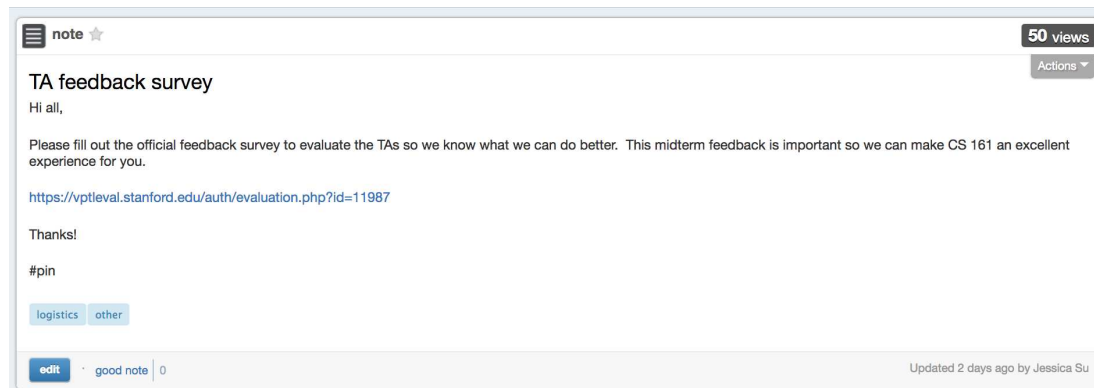


Lecture 10

Finding strongly connected components

Announcements

- HW4 due Monday
- The TAs posted a feedback form on Piazza:



- MIDTERM on Wednesday 5/10
 - Covers material up through today
 - In class, in this room
 - Practice midterms on the website
 - Disclaimer: these are from past quarters and may not cover exactly the same material. But they are a good approximation.

Last time

- Breadth-first and depth-first search
- Plus, applications!
 - Topological sorting
 - In-order traversal of BSTs
 - Shortest path in unweighted graphs
 - Testing bipartite-ness
- The key was paying attention to the structure of the tree that these search algorithms implicitly build.

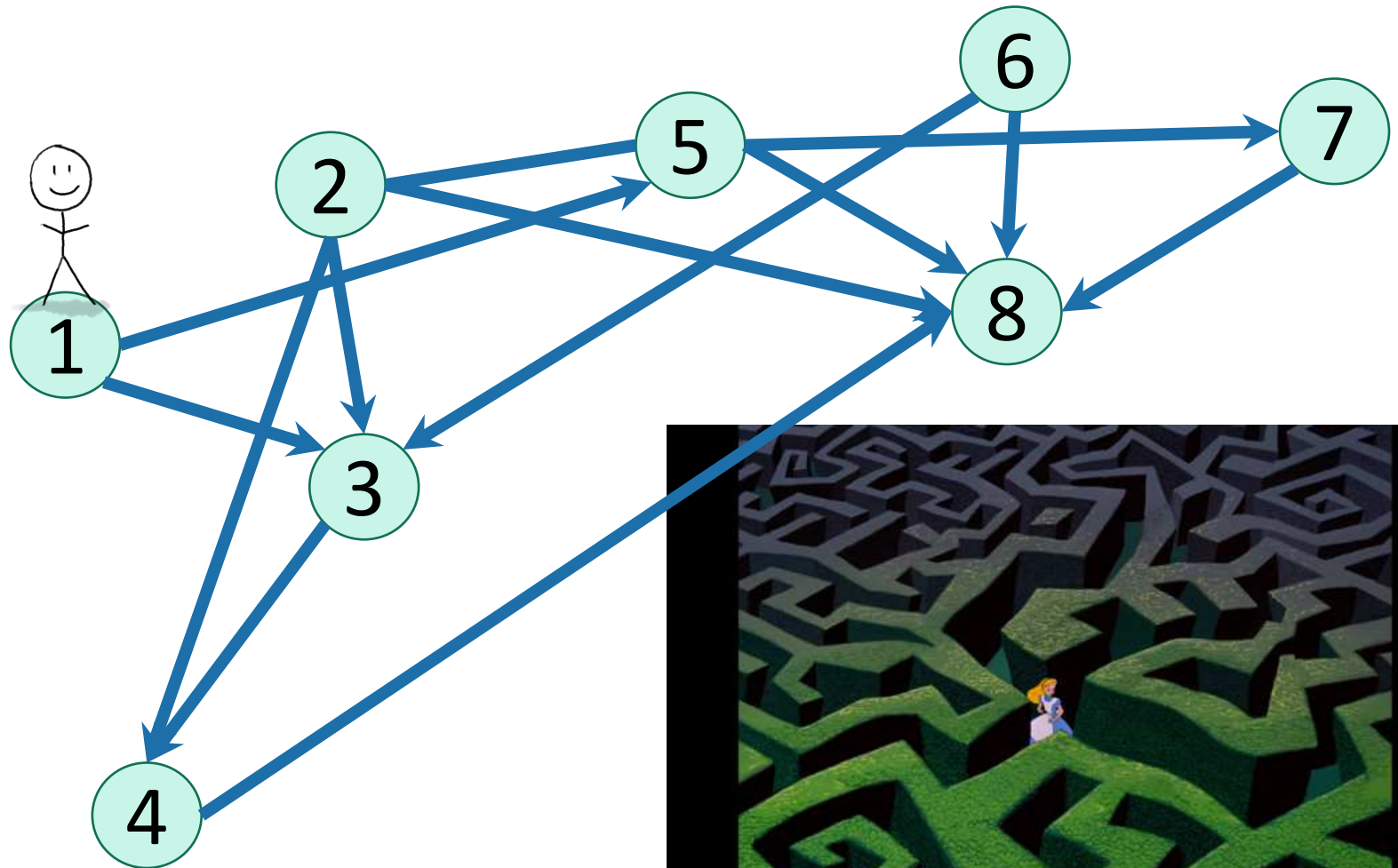
Today

- One more application:
 - Finding **strongly connected components**
- This is a tricky problem that you **(or at least I)** **probably couldn't solve** without the techniques from last lecture.
 - **Moral of the story: CS161 FTW**
- But first! Let's briefly recap DFS...

Recall: DFS

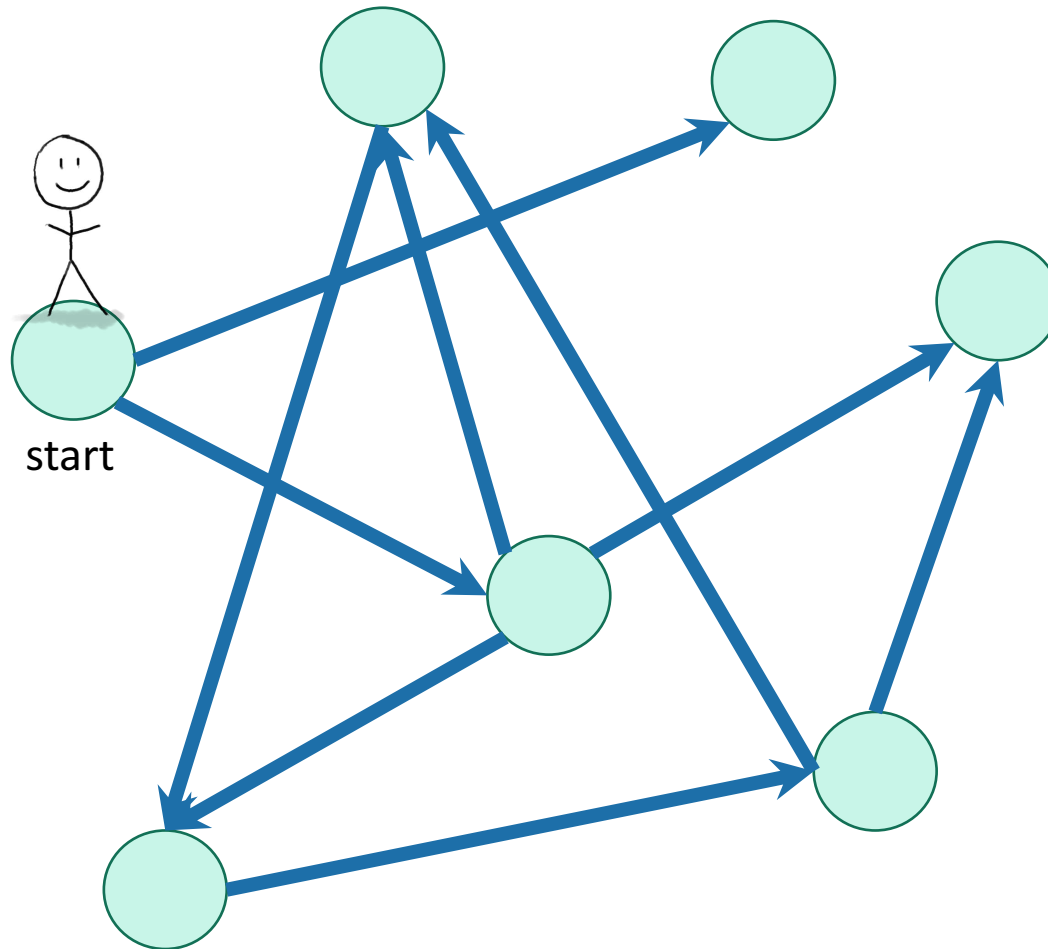
It's how you'd explore a labyrinth with chalk and a piece of string.

Today, all graphs are **directed**!
Check that the things we did
on Monday still all work!



Depth First Search

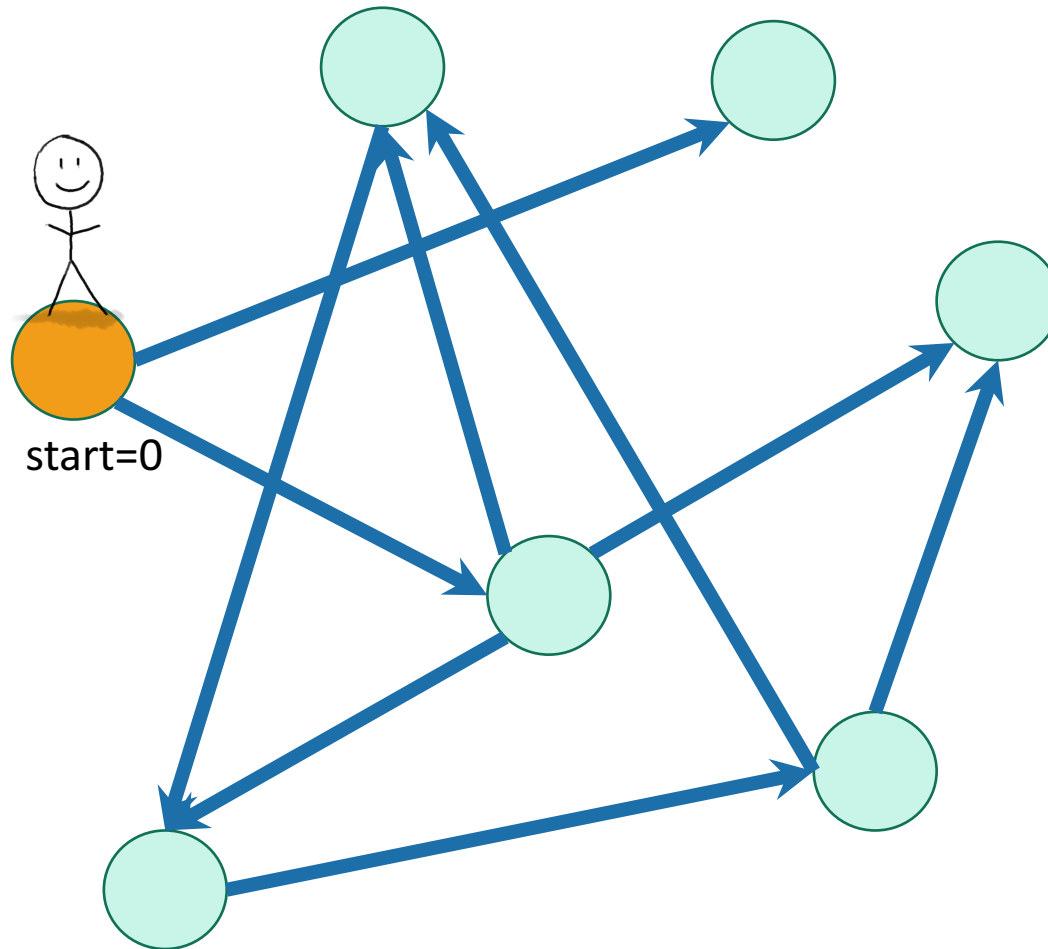
Exploring a maze with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

This is the same picture we had Monday, except I've directed all of the edges. Notice that there **ARE** cycles.

Exploring a labyrinth with chalk and a piece of string



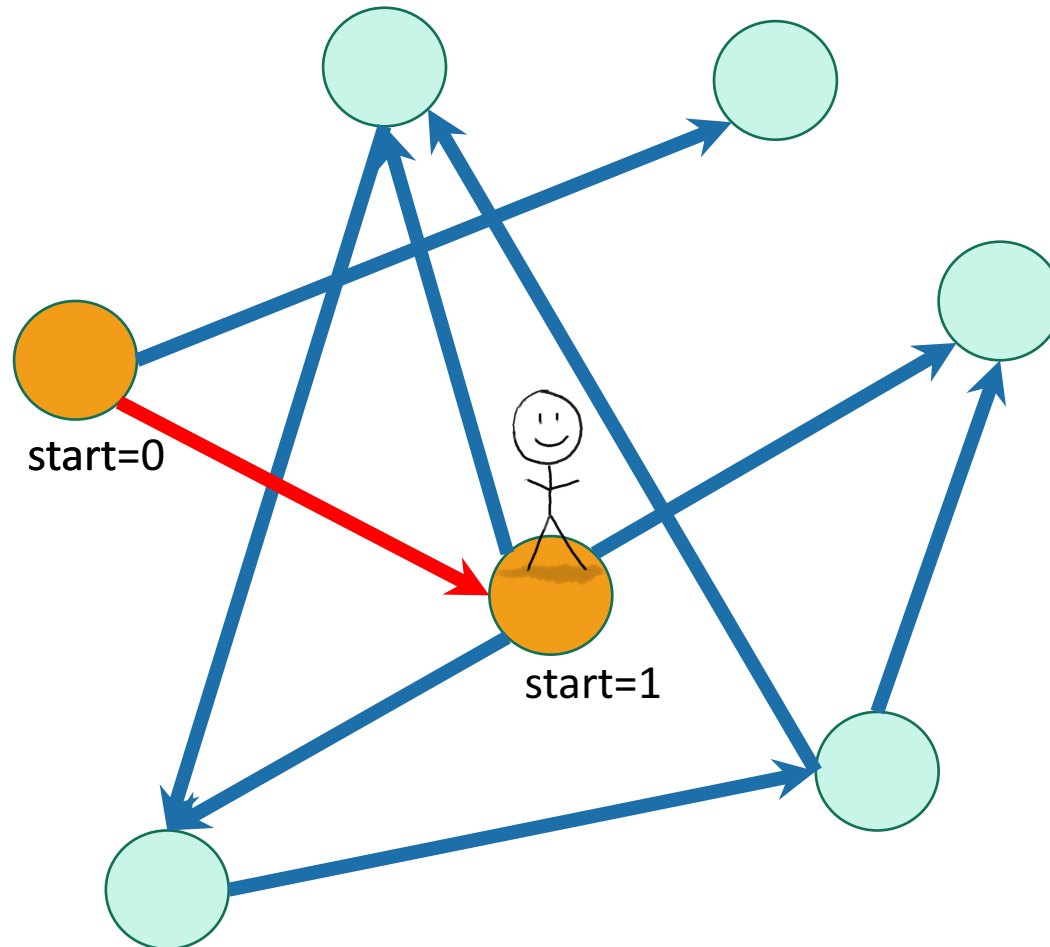
- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.



Recall we also keep track of **start** and **finish** times for every node.

Depth First Search

Exploring a labyrinth with chalk and a piece of string



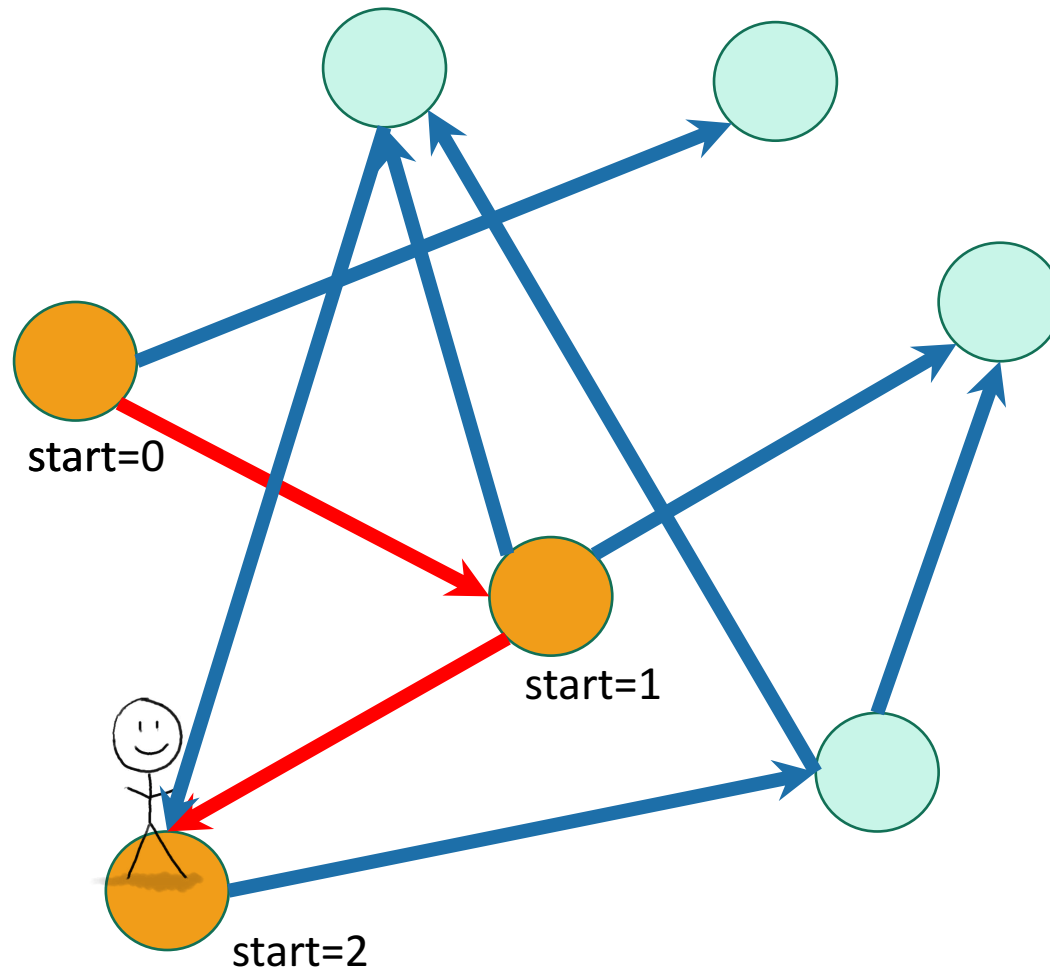
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.



Recall we also keep track of **start** and **finish** times for every node.

Depth First Search

Exploring a labyrinth with chalk and a piece of string



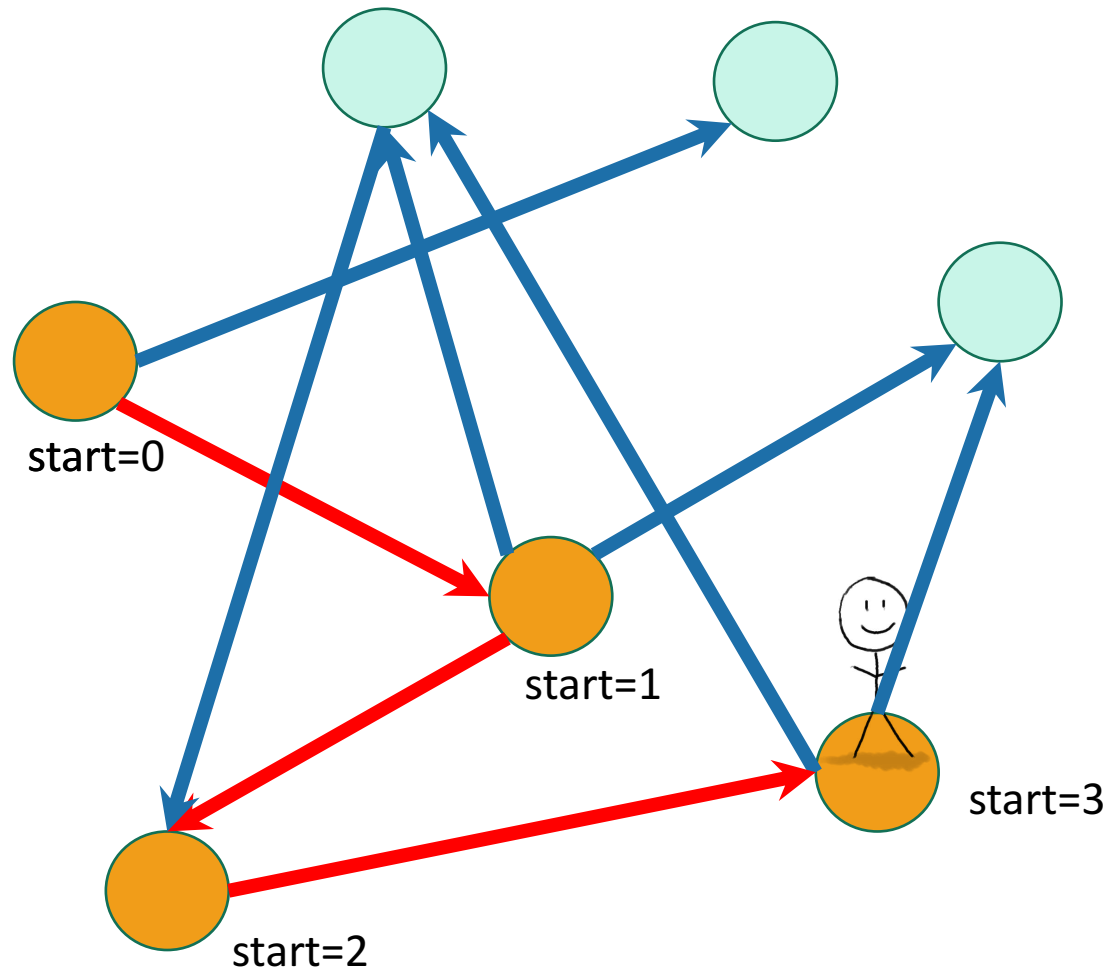
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.



Recall we also keep track of **start** and **finish** times for every node.

Depth First Search

Exploring a labyrinth with chalk and a piece of string



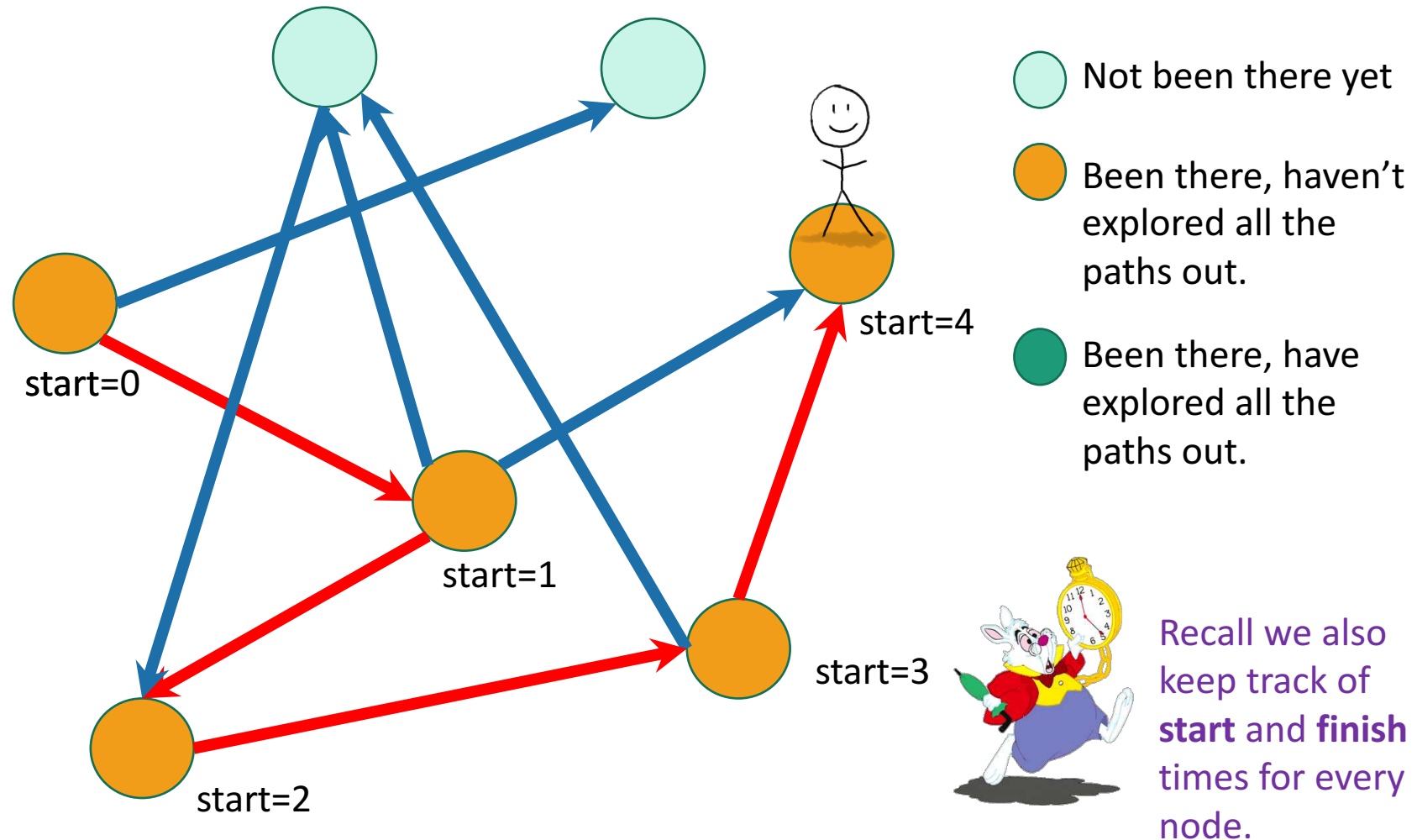
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.



Recall we also keep track of **start** and **finish** times for every node.

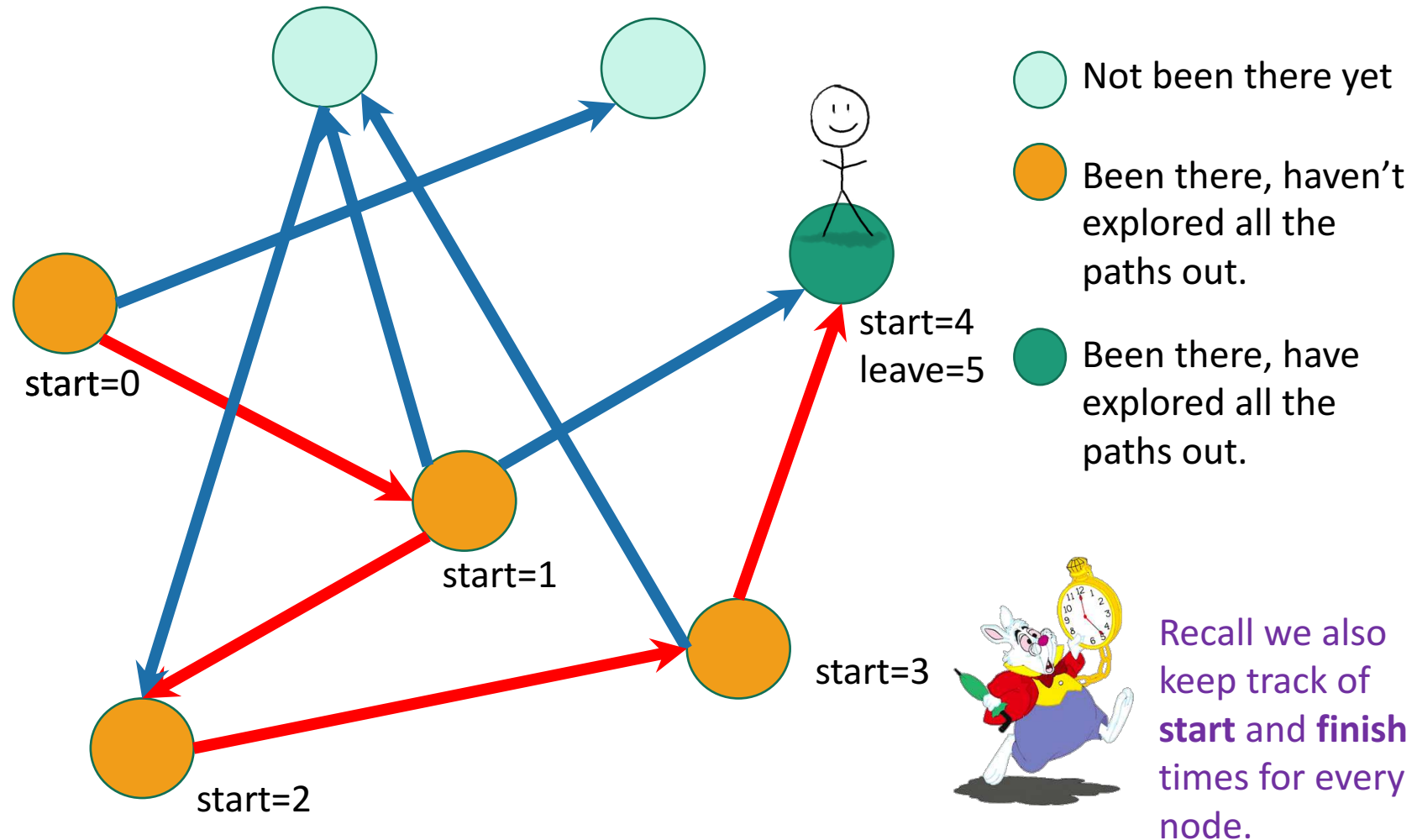
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

Exploring a labyrinth with chalk and a piece of string

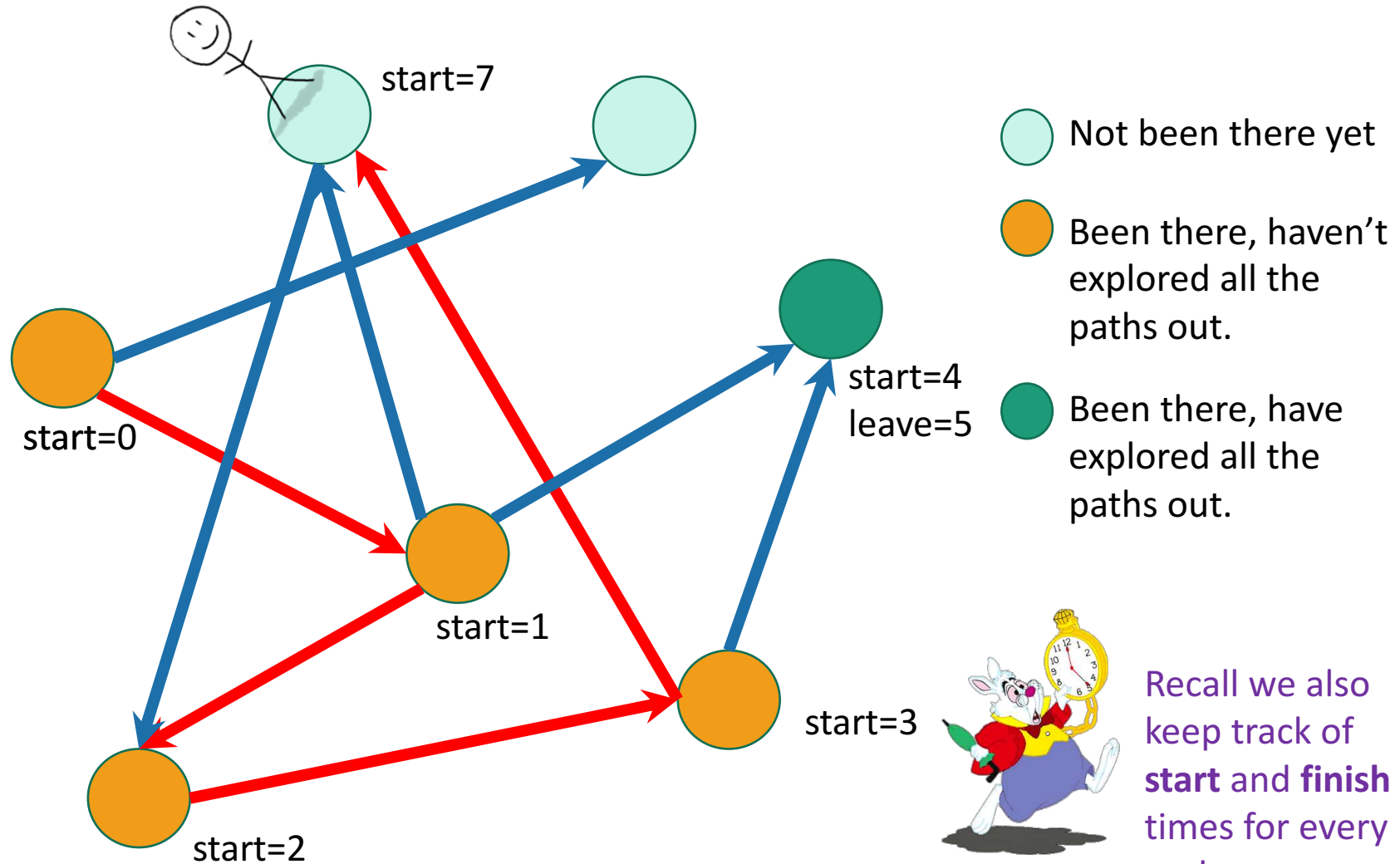


Exploring a labyrinth with chalk and a piece of string



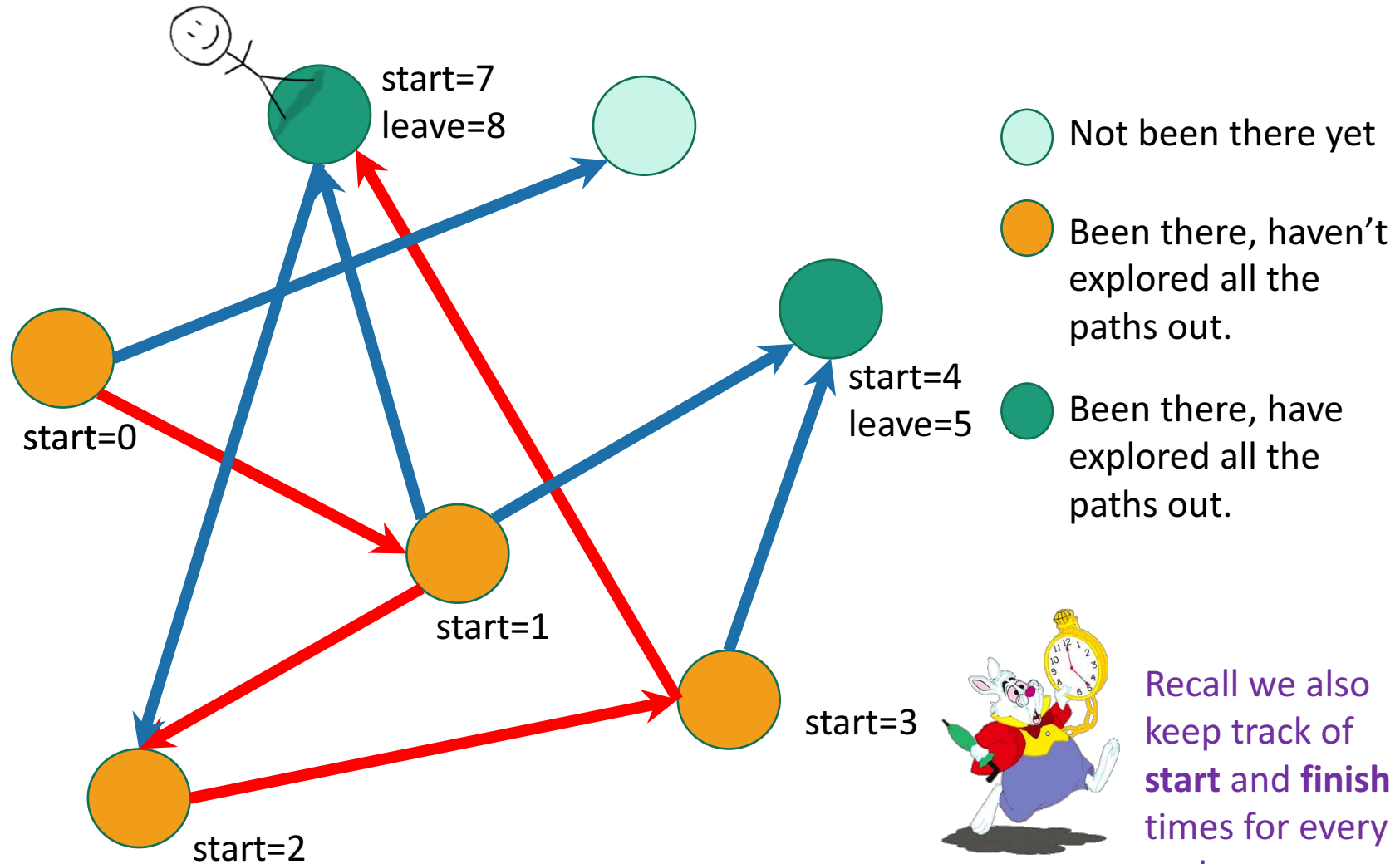
Depth First Search

Exploring a labyrinth with chalk and a piece of string



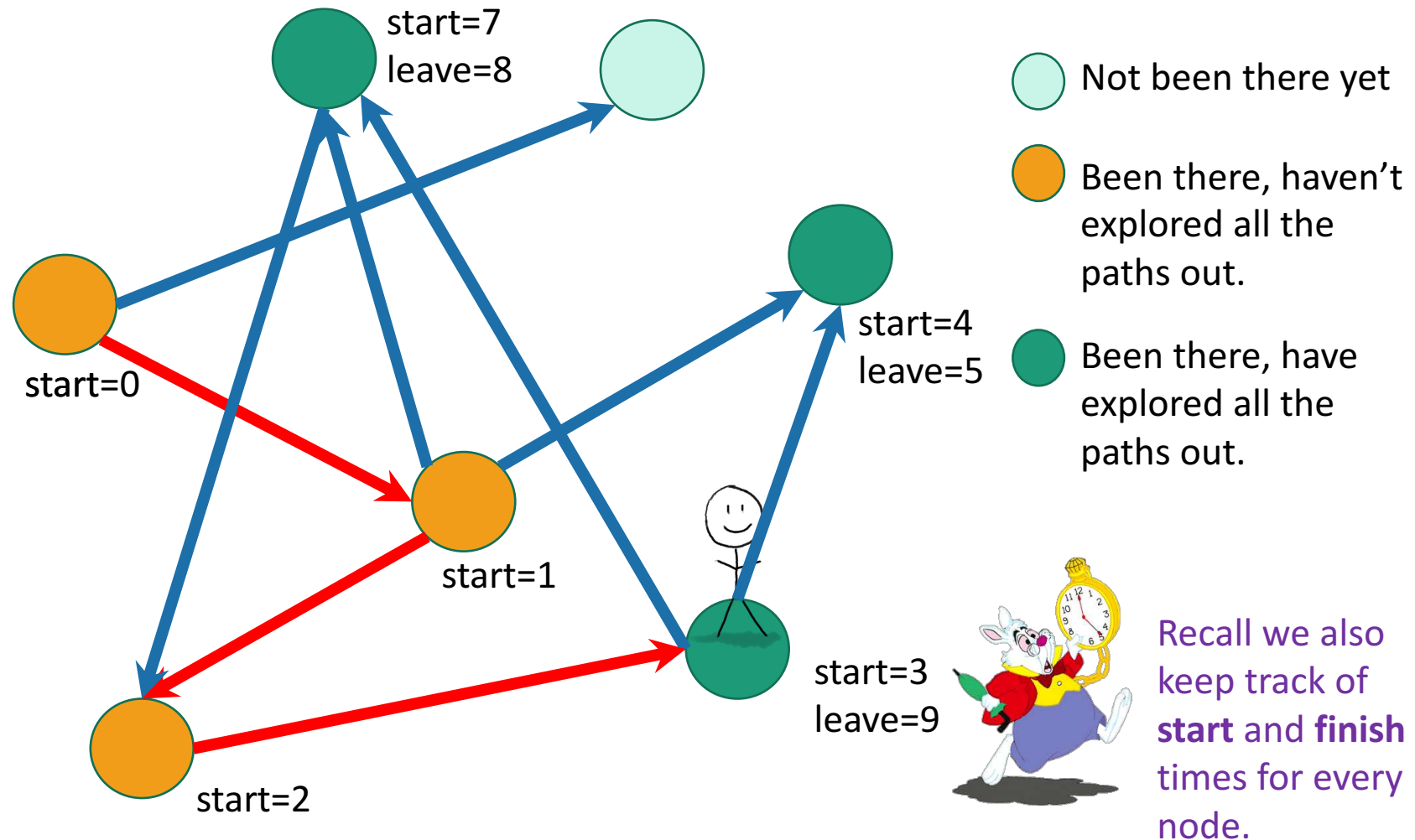
Depth First Search

Exploring a labyrinth with chalk and a piece of string



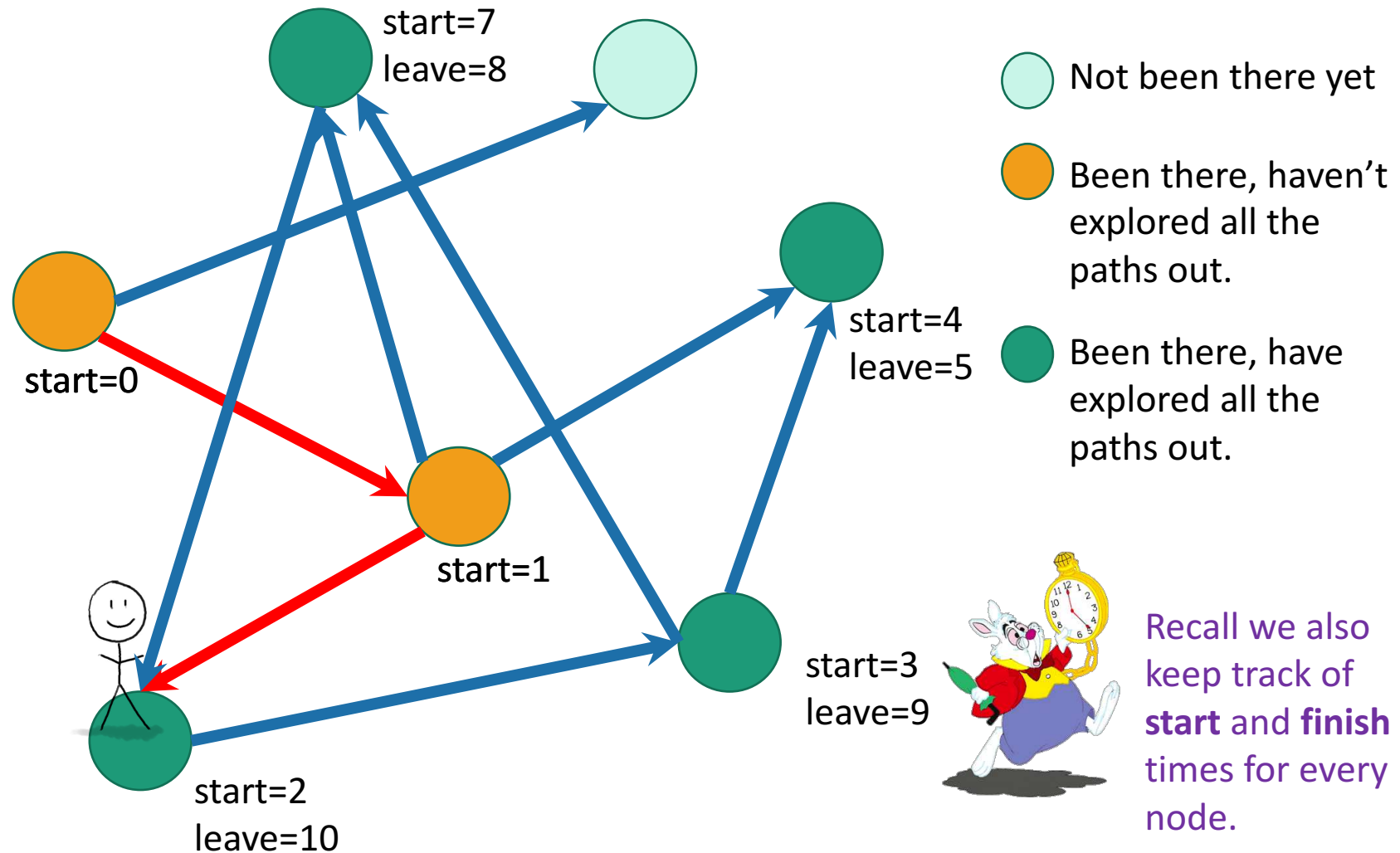
Depth First Search

Exploring a labyrinth with chalk and a piece of string



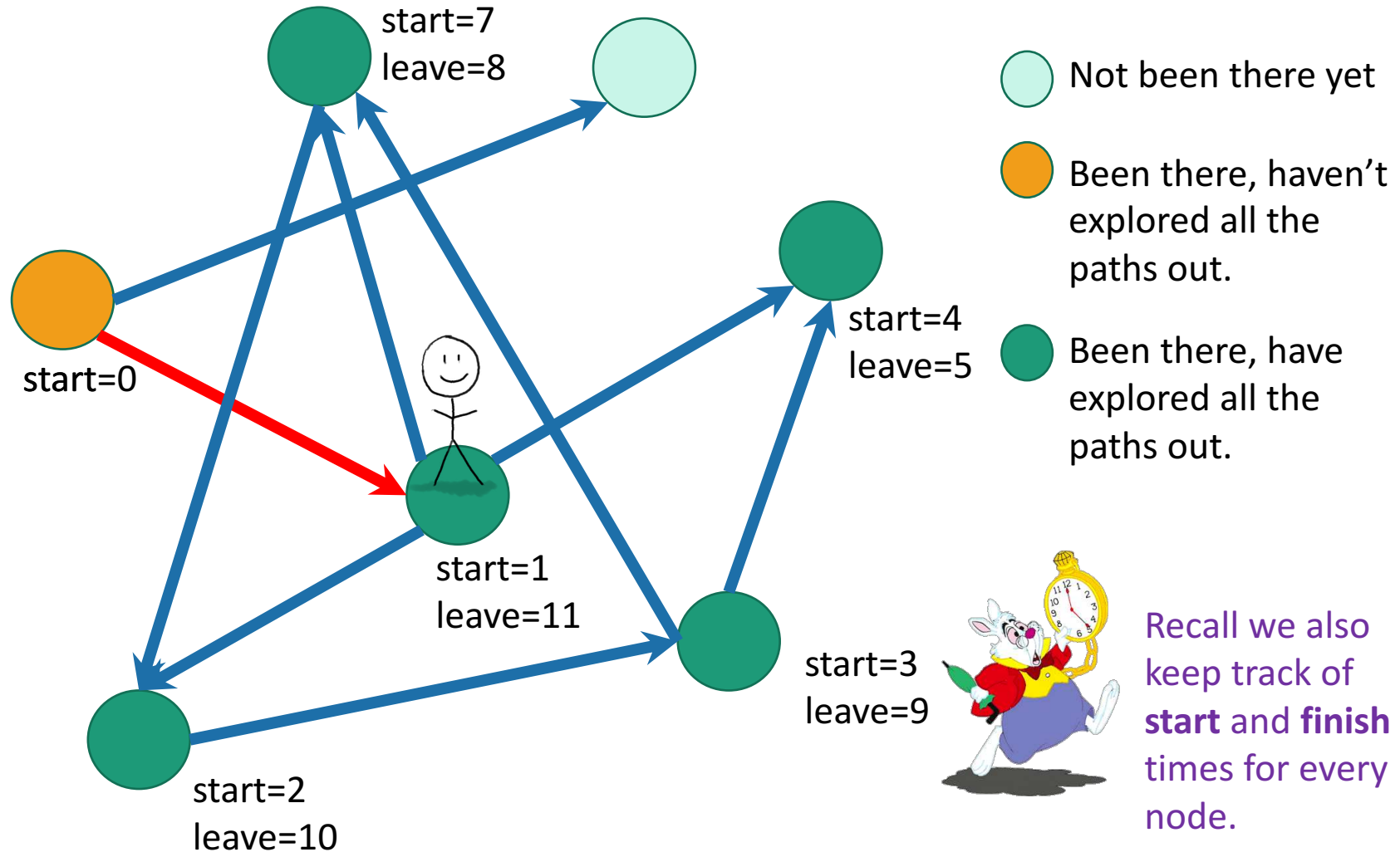
Depth First Search

Exploring a labyrinth with chalk and a piece of string



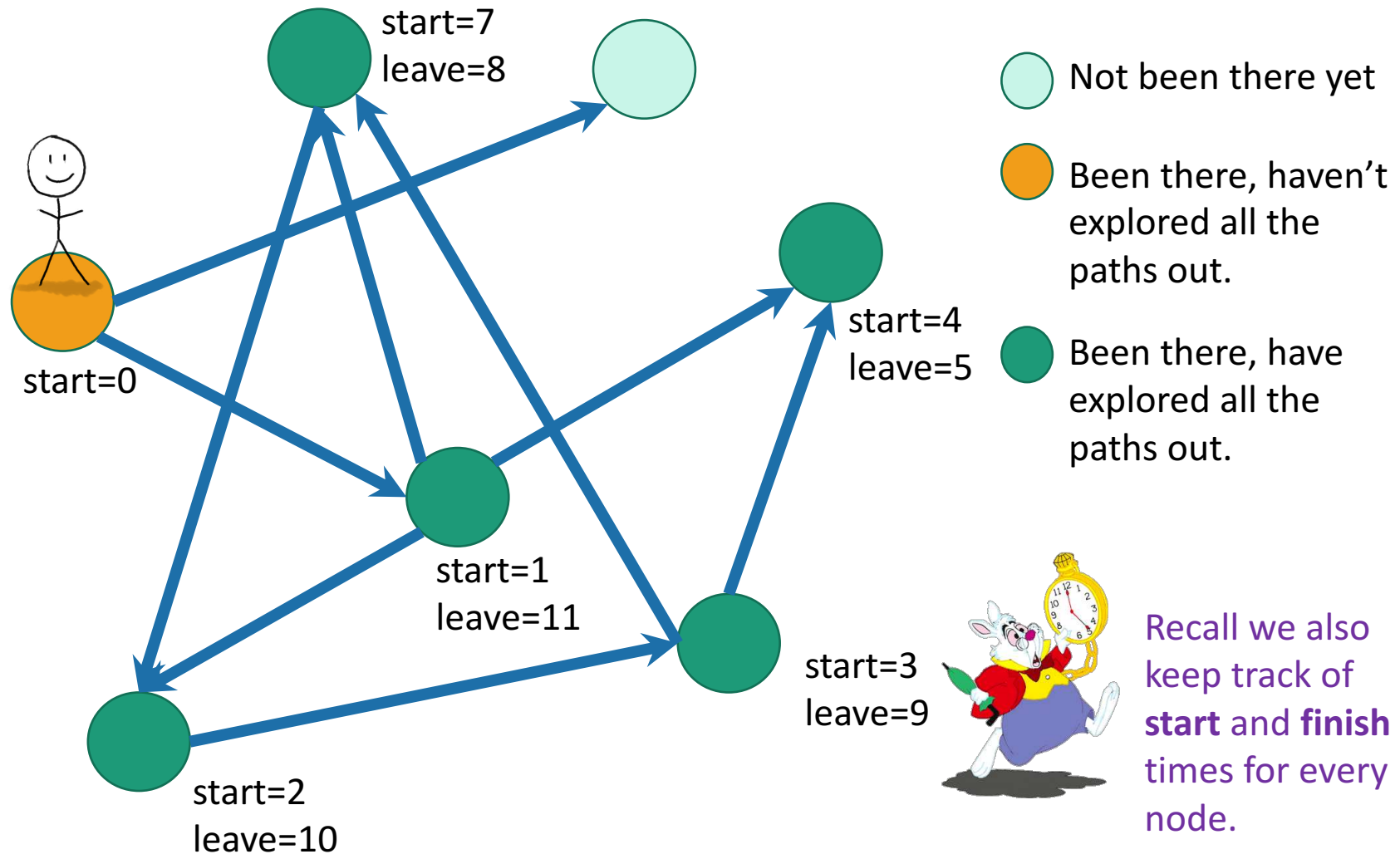
Depth First Search

Exploring a labyrinth with chalk and a piece of string



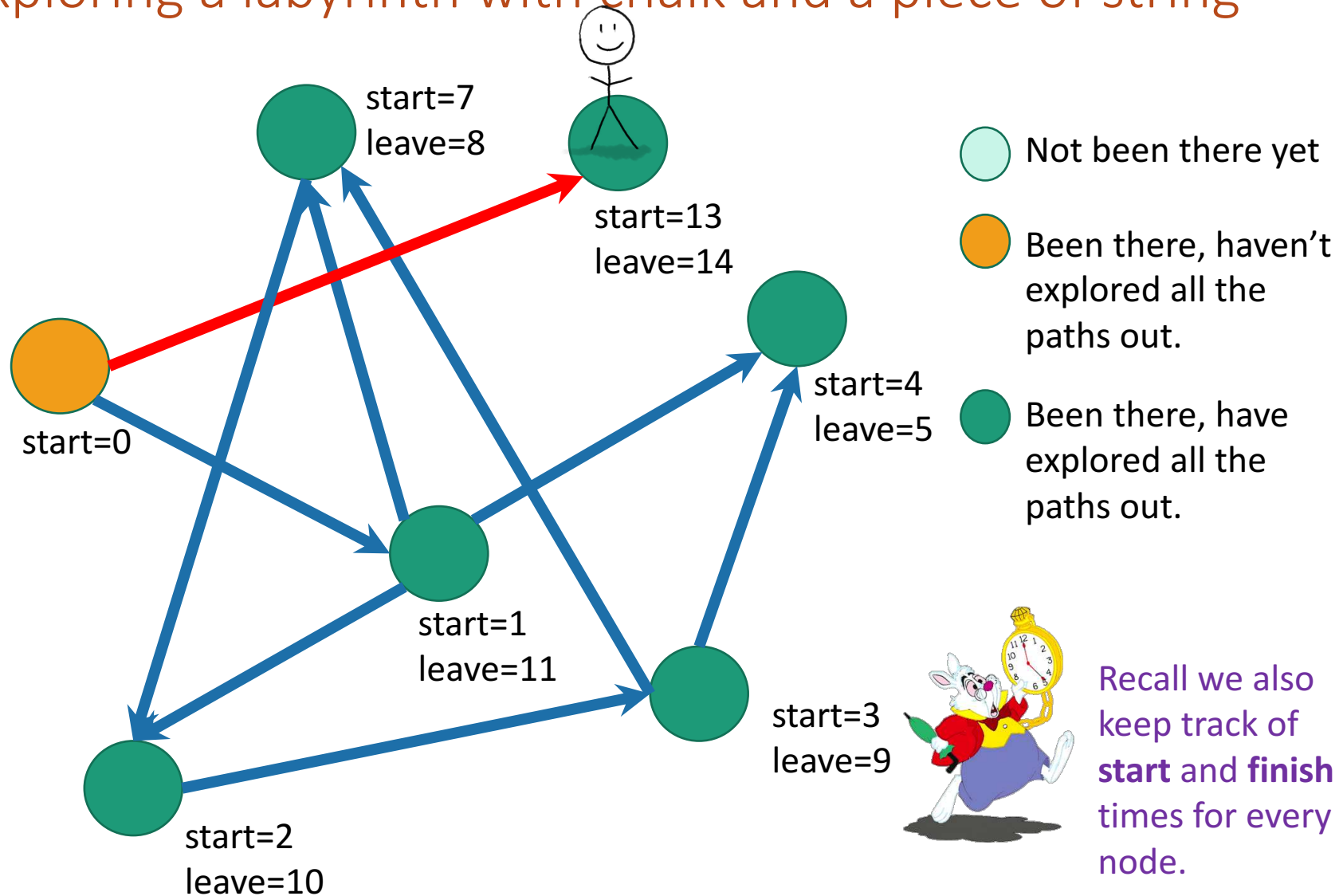
Depth First Search

Exploring a labyrinth with chalk and a piece of string



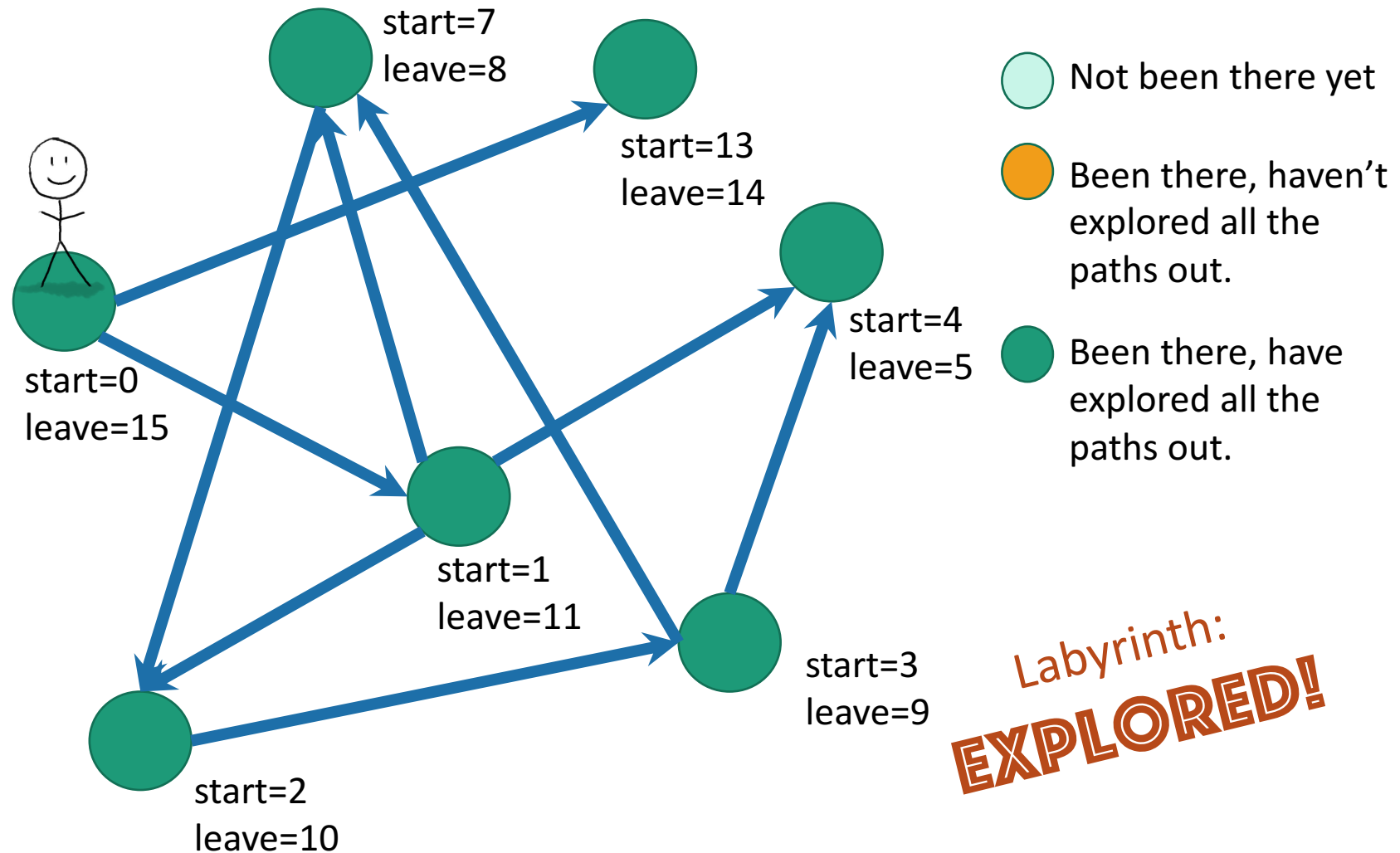
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

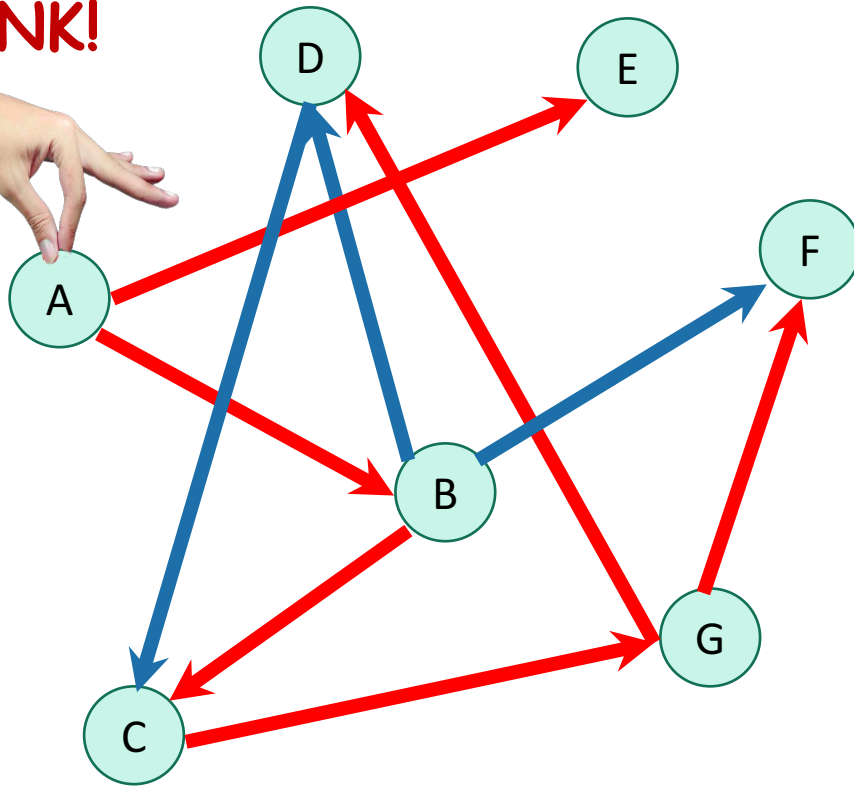
Exploring a labyrinth with chalk and a piece of string



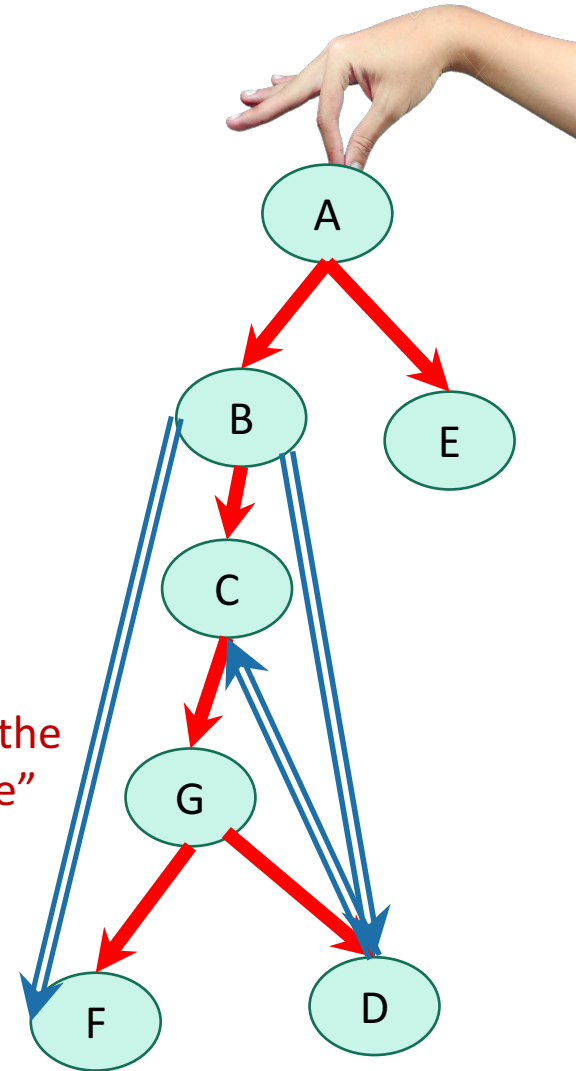
Depth first search

implicitly creates a tree on everything you can reach

YOINK!

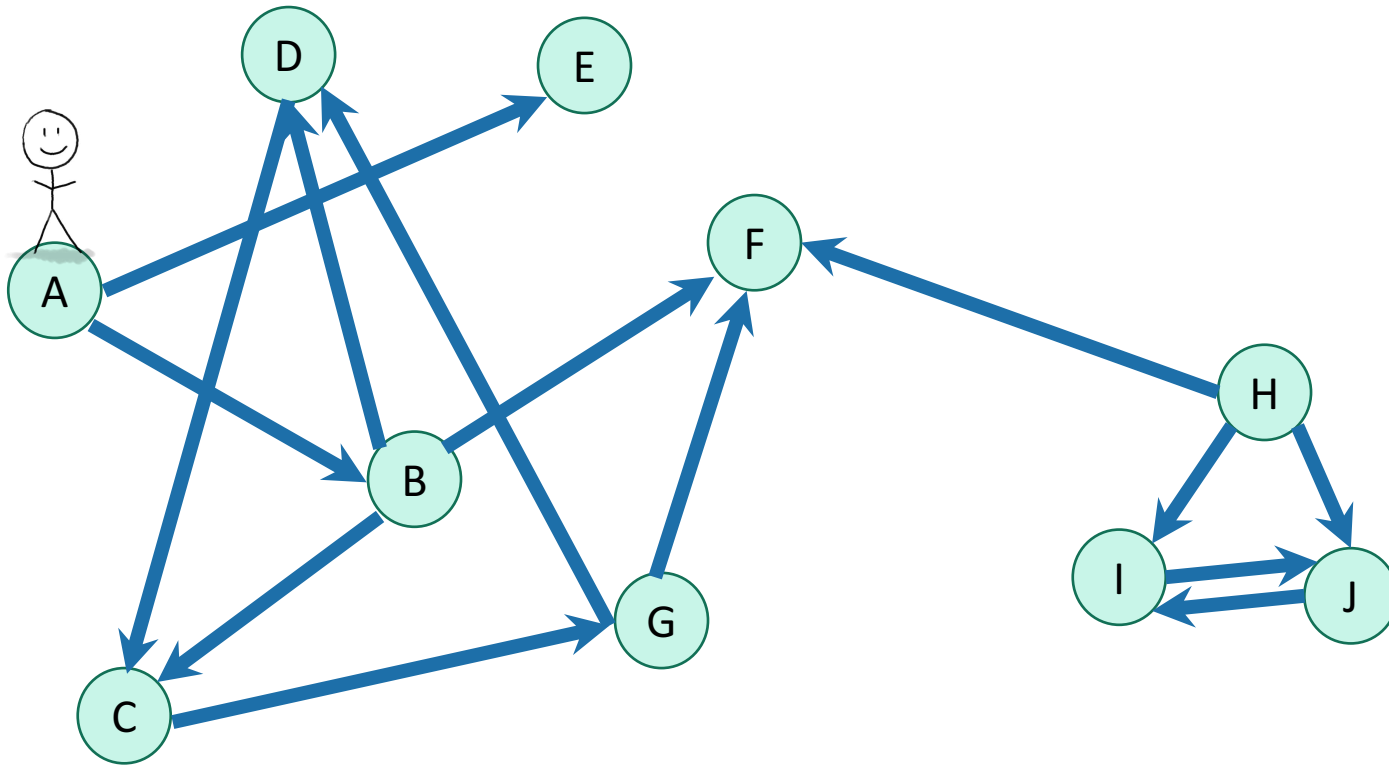


Call this the
"DFS tree"



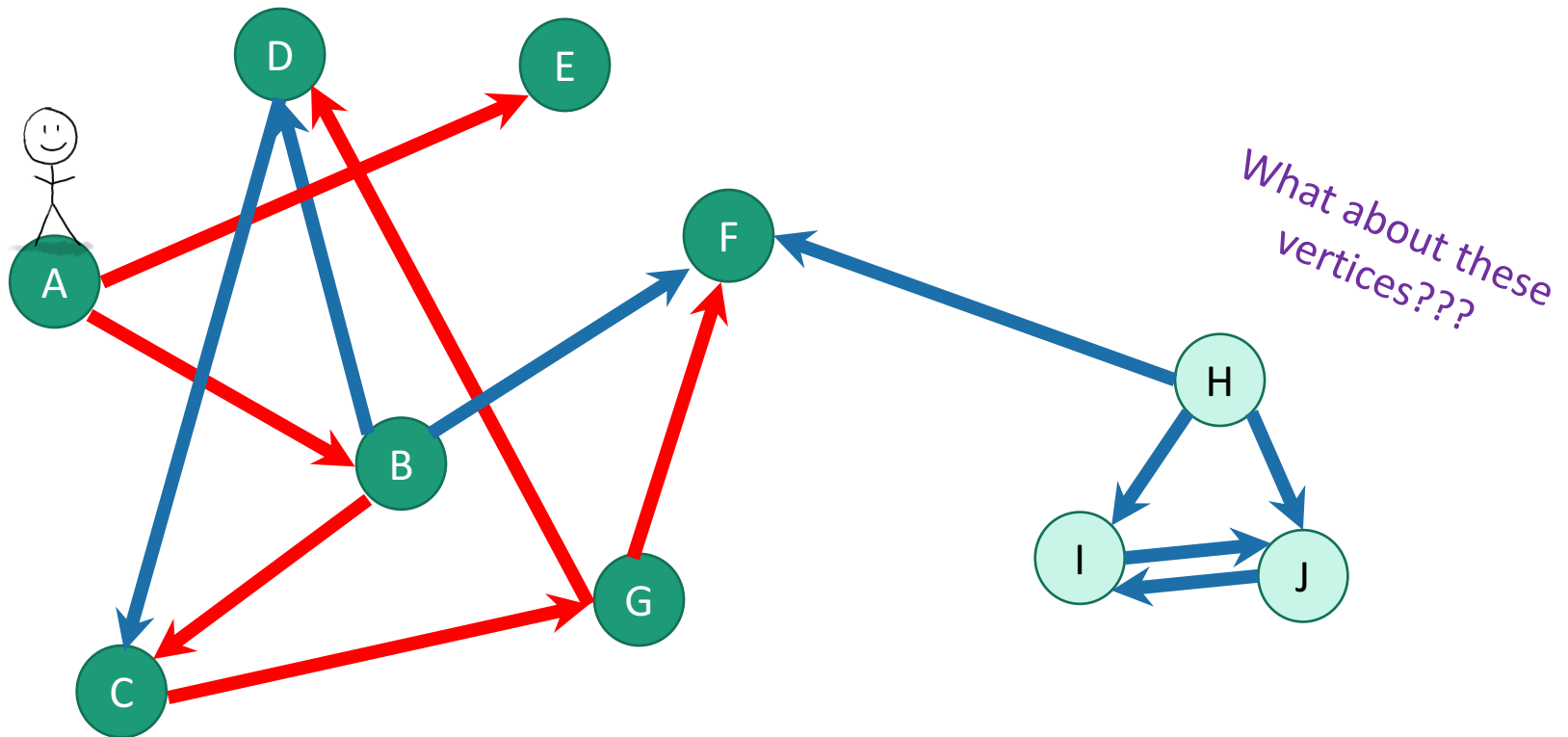
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



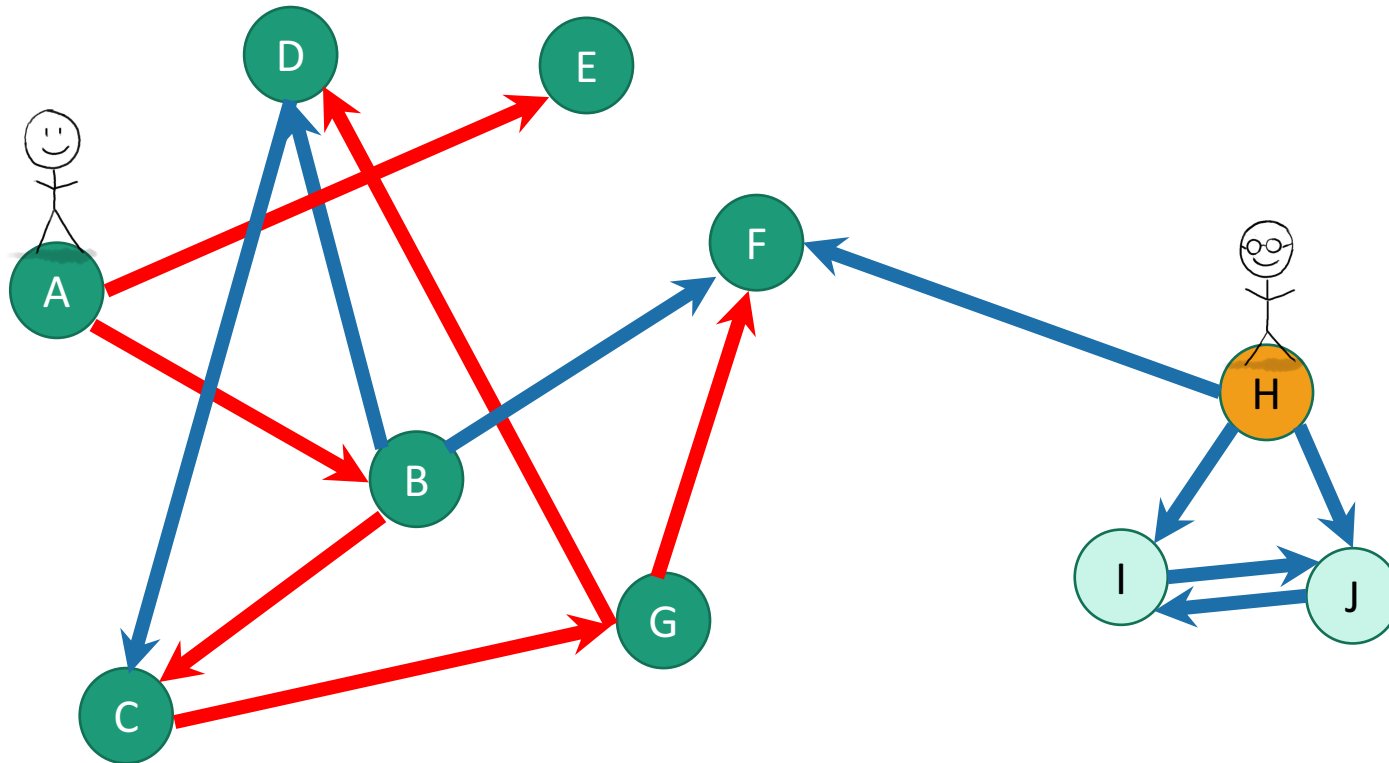
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



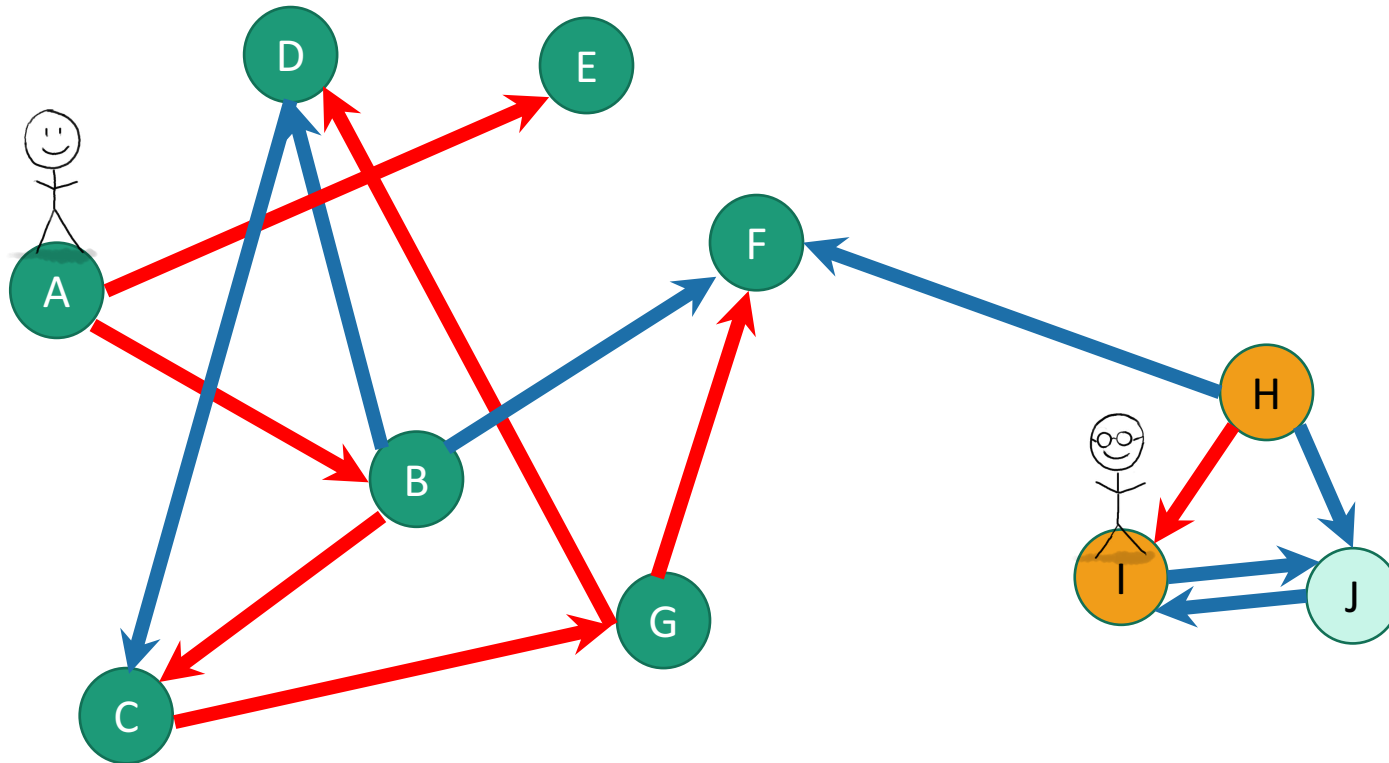
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



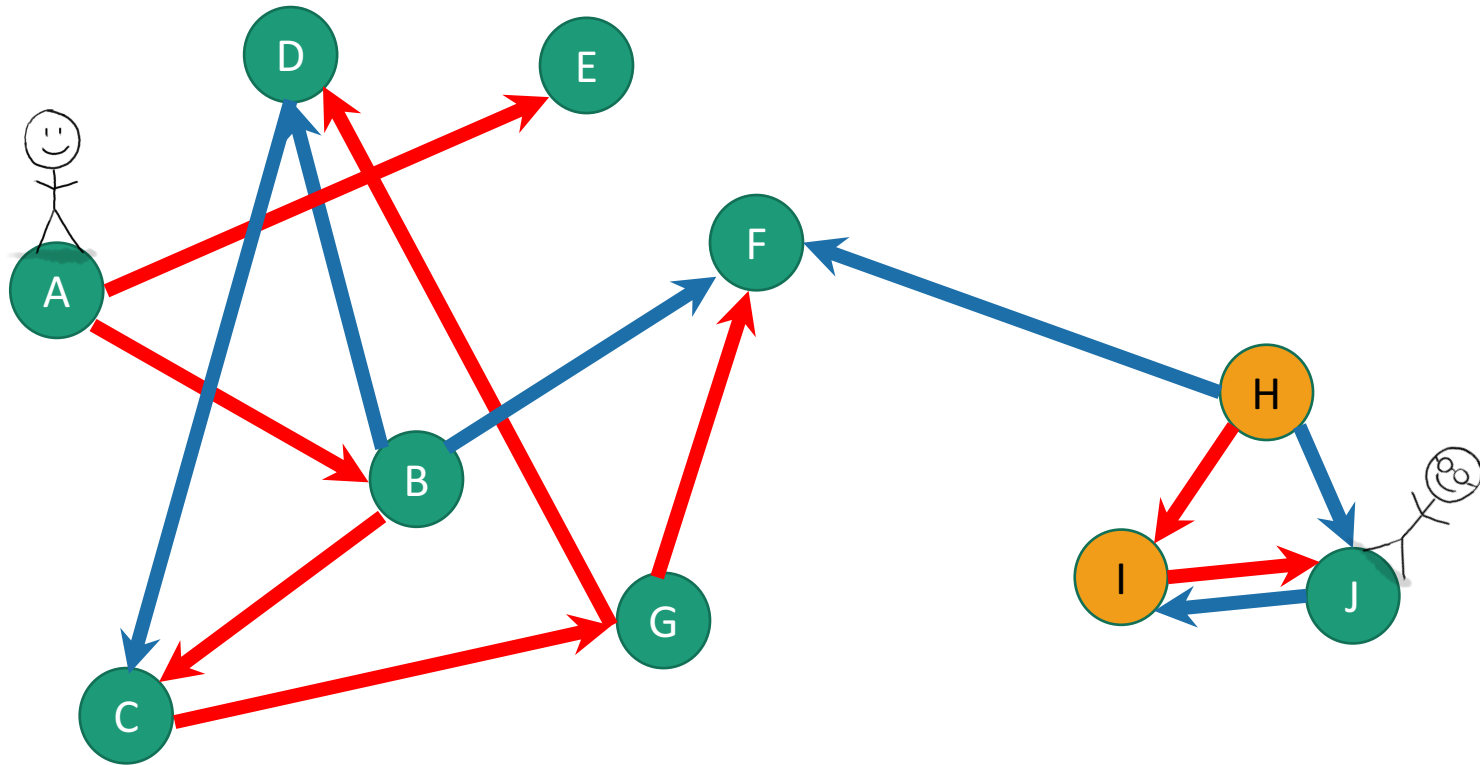
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



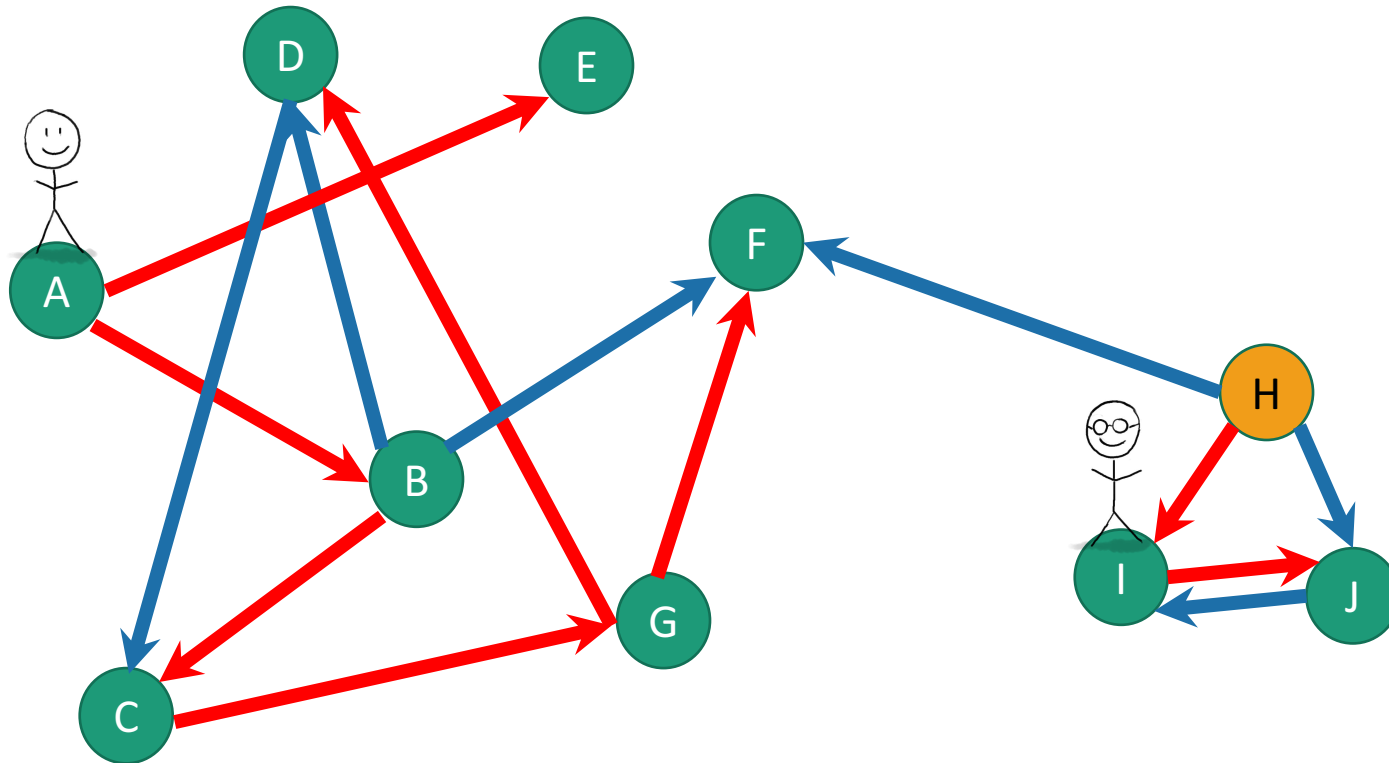
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



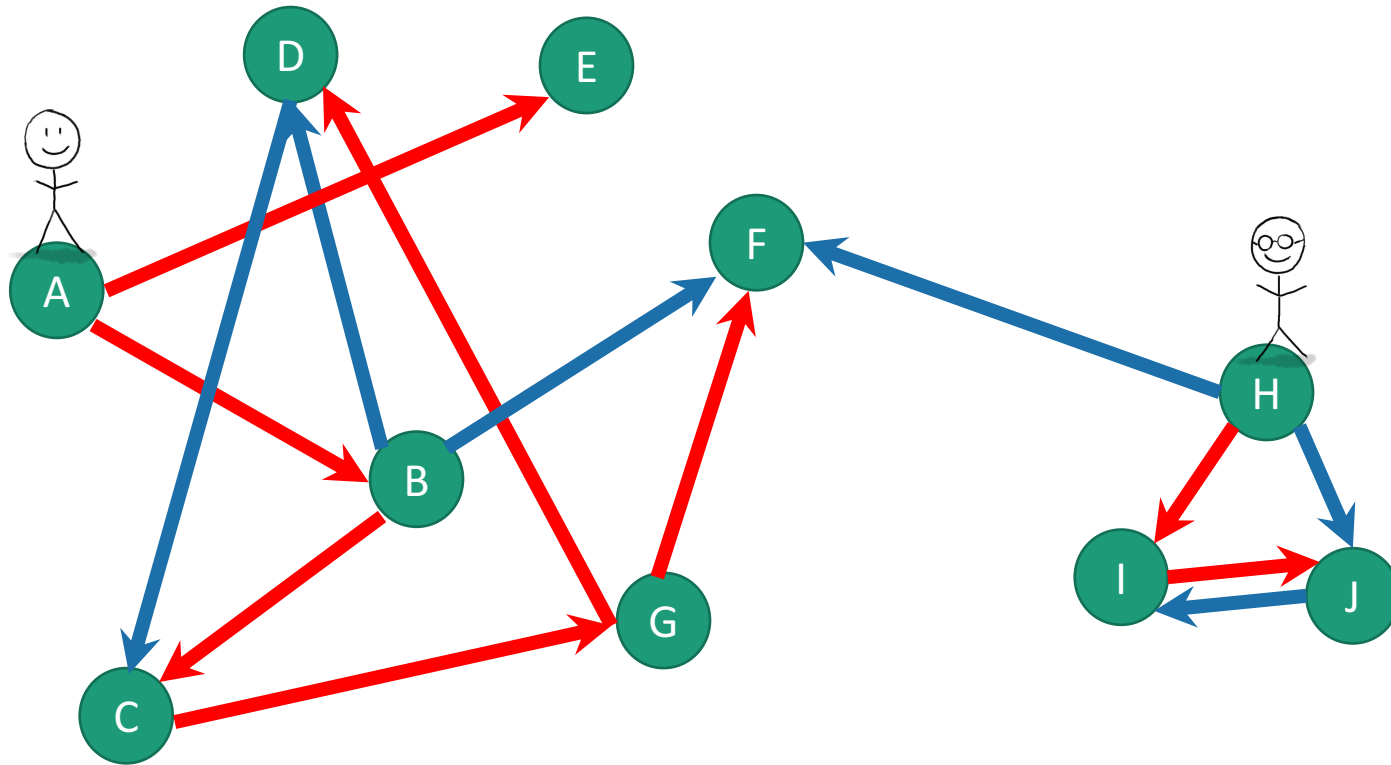
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



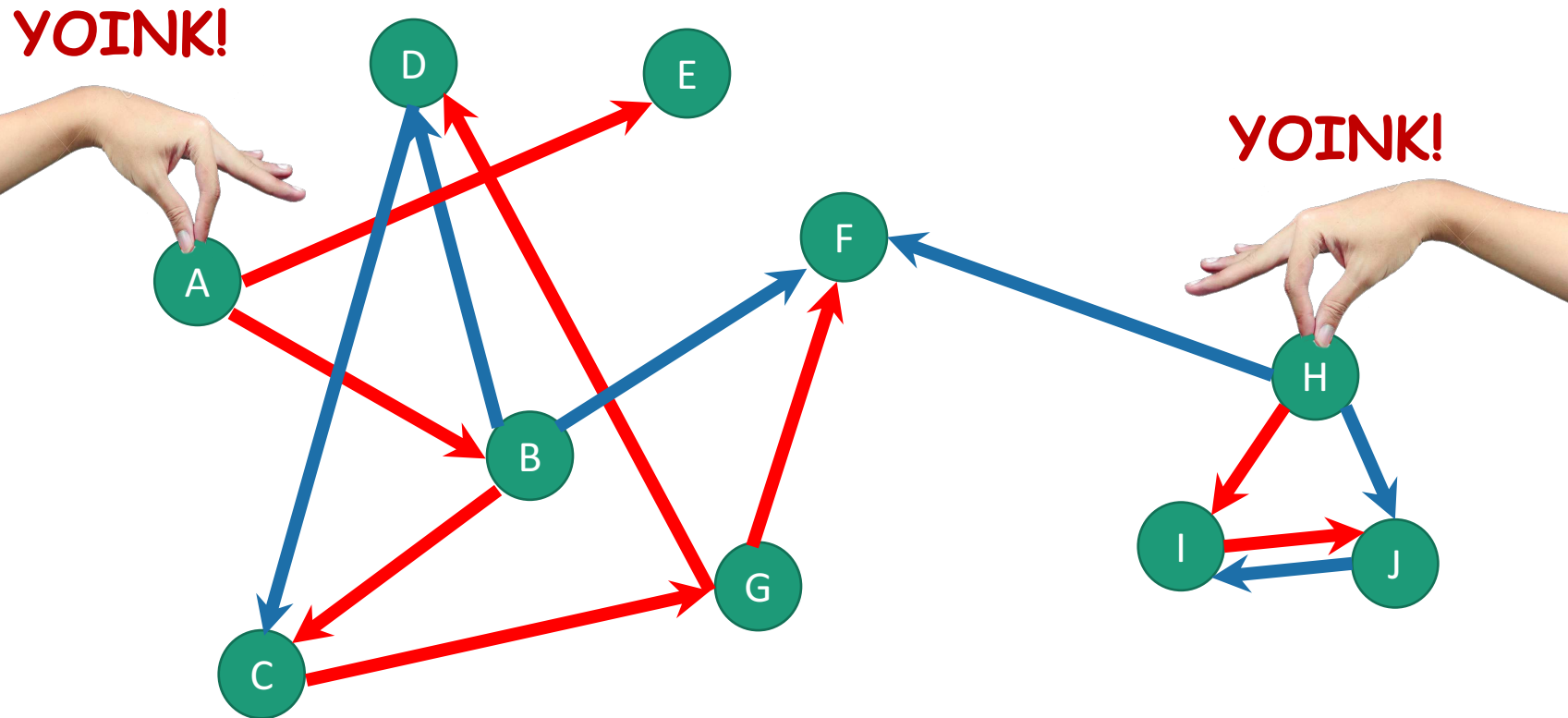
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



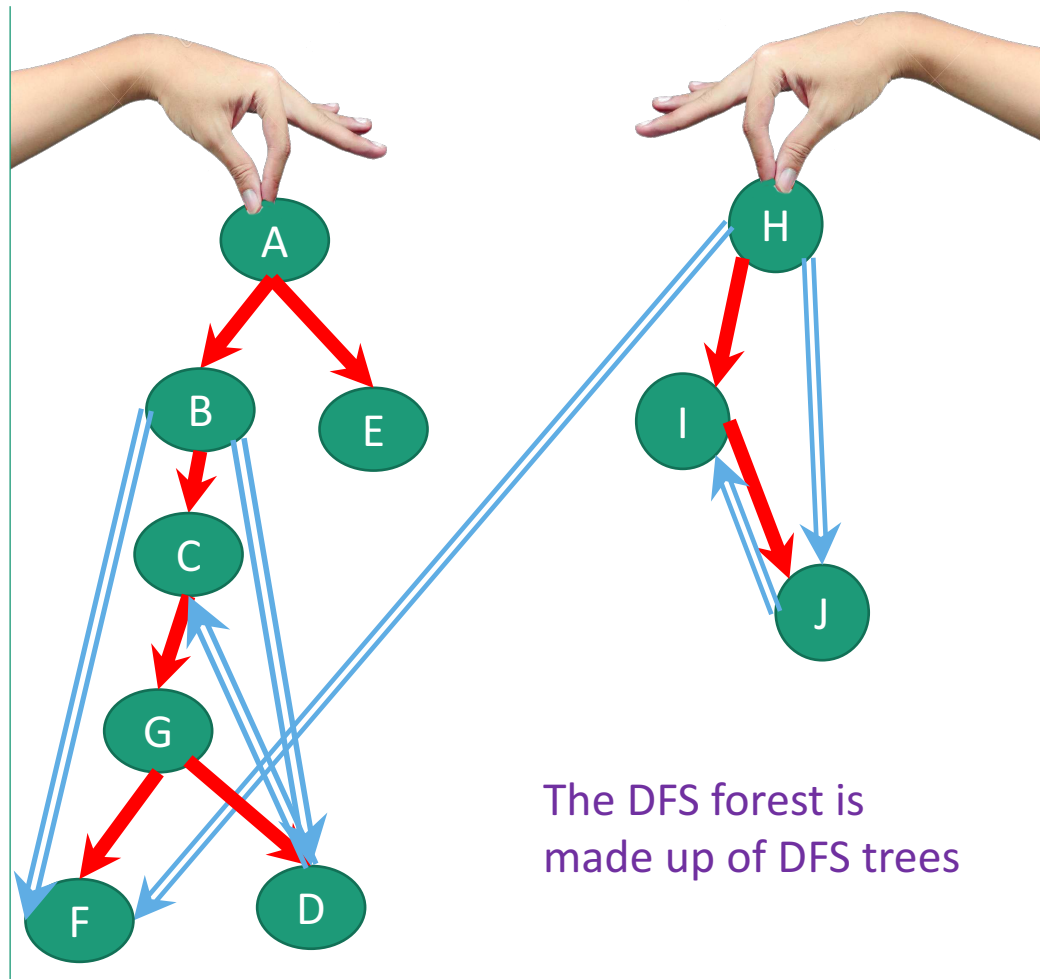
When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



How long does this take?

- $O(m + n)$... why?
- Ollie the over-achieving ostrich told you to figure this out on your own, but let's go over it.
- The pseudocode looked like:
 - **DFS(w):**
 - Do some bookkeeping stuff
 - For v in w .neighbors:
 - DFS(v)
 - Do some more bookkeeping stuff
- The bookkeeping stuff takes time $O(1)$
- Then there's that loop, time $O(\deg(w))$ in DFS(w).
- Total amount is $\sum_{w \in V} O(1) + O(\deg(w)) = O(n + m)$.



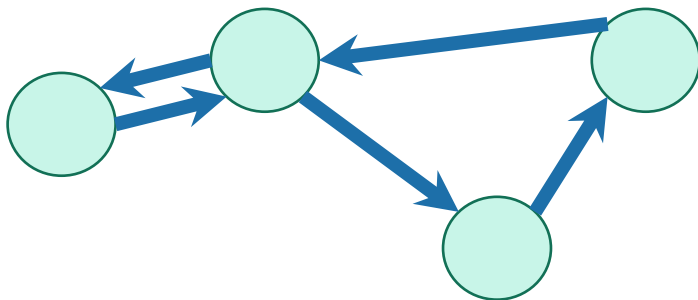
I claimed that to do just one “reachable component,” the time was $O(m)$. **That's true.** Why?

Enough of review

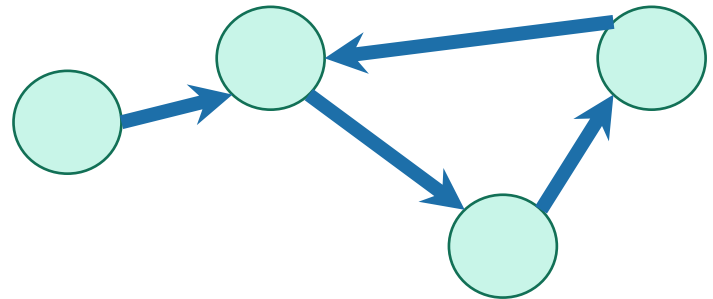
Strongly connected components

Strongly connected components

- A directed graph $G = (V, E)$ is **strongly connected** if:
- for all v, w in V :
 - there is a path **from v to w** and
 - there is a path **from w to v** .



strongly connected

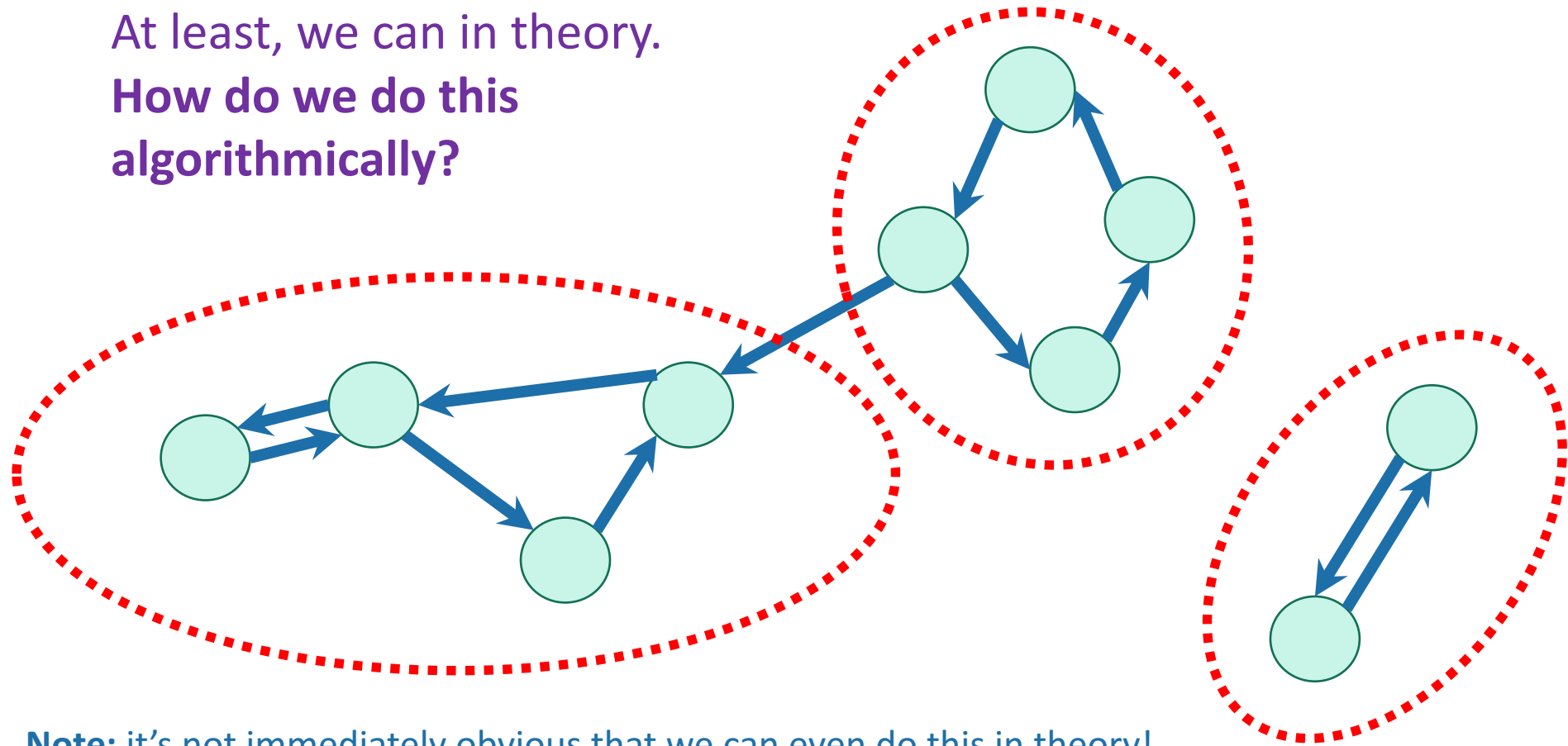


not strongly connected

We can decompose a graph into **strongly connected components** (SCCs)

At least, we can in theory.

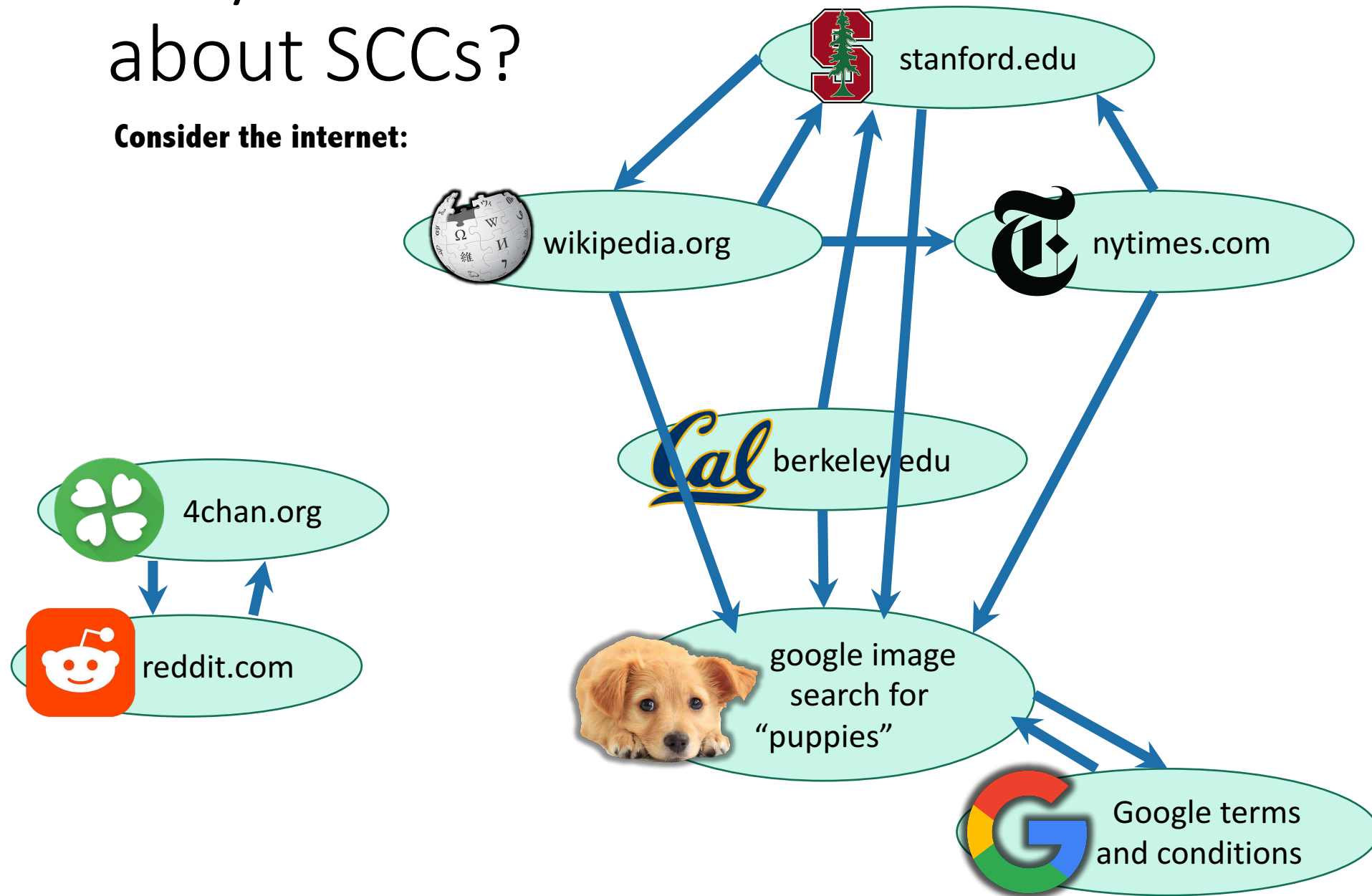
How do we do this algorithmically?



Note: it's not immediately obvious that we can even do this in theory!
The reason why is because “**two vertices are reachable from each other**” is an **equivalence relation**, and the SCCs are **equivalence classes**.

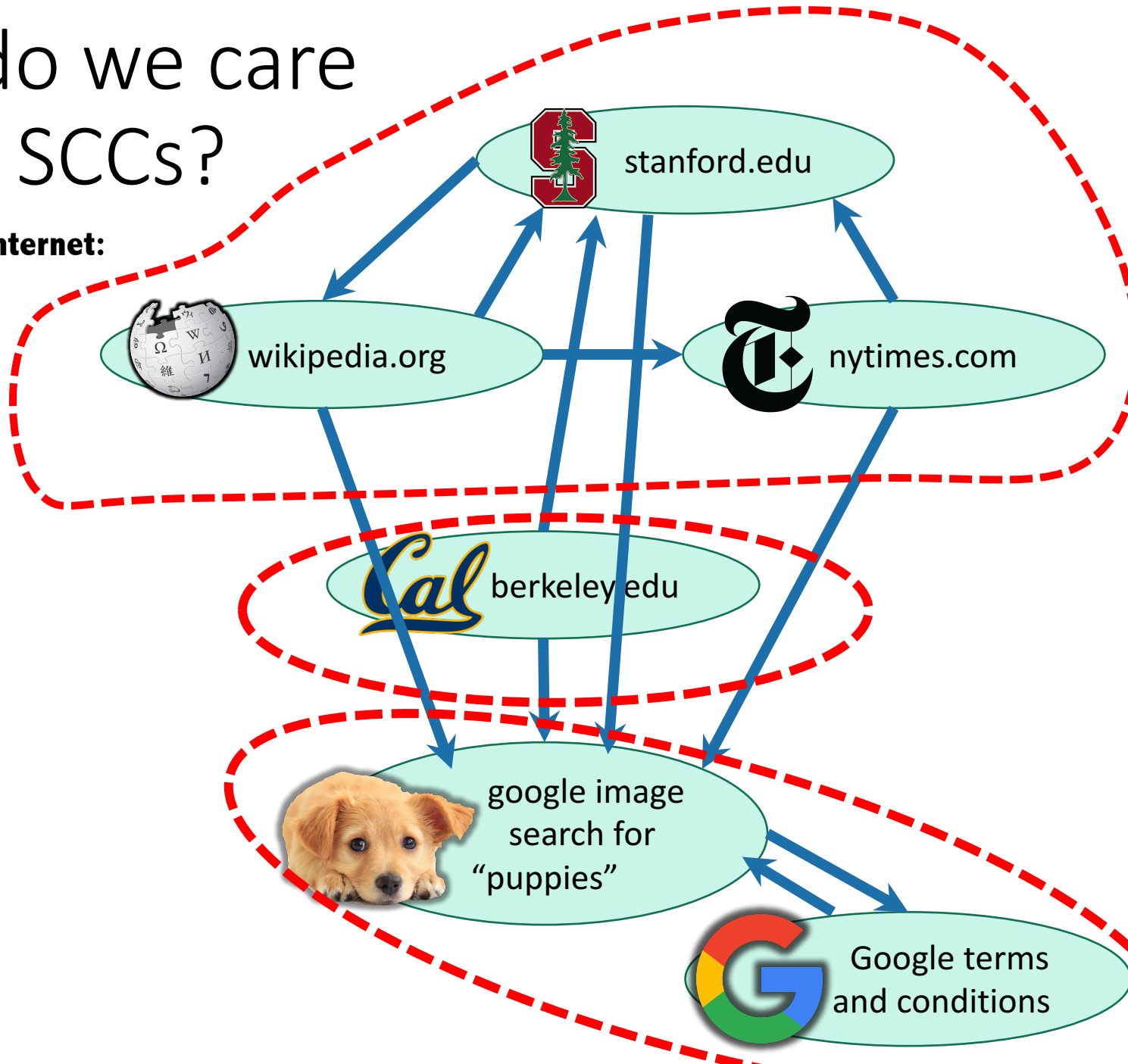
Why do we care about SCCs?

Consider the internet:



Why do we care about SCCs?

Consider the internet:



What are the SCCs of the internet?

- In real life, turns out there's one “giant” one.
 - and then a bunch of tendrils.
- More generally:
 - Strongly connected components tell you about **communities**.
 - Lots of graph algorithms only make sense on strongly connected components, so **sometimes we want to find them for other reasons**.
- This is true not just in this internet/interaction graphs.
 - I was talking to an economist the other day who has to first break up his labor market data into SCCs in order to make sense of it.

How to find SCCs?

Try 1:

- Consider all possible decompositions and check.

Try 2:

- For each pair (u,v) ,
 - use DFS to find if there are paths u to v and v to u .
- Aggregate accordingly.
- Running time:
 - This takes time at least $O(n^3)$ to actually do it this way
 - Maybe you can get away with $O(n^2)$...
 - But definitely no better than $O(n^2)$.

What if I told you I could do it in time $O(\log(n))$?

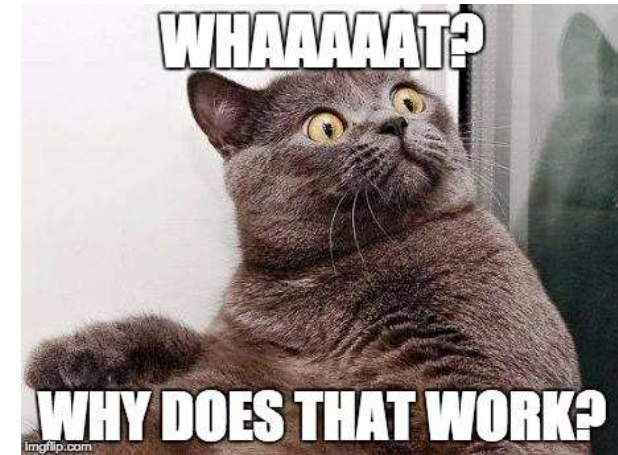
Two legit responses:

- Nuh-uh!
- In what model of computation?

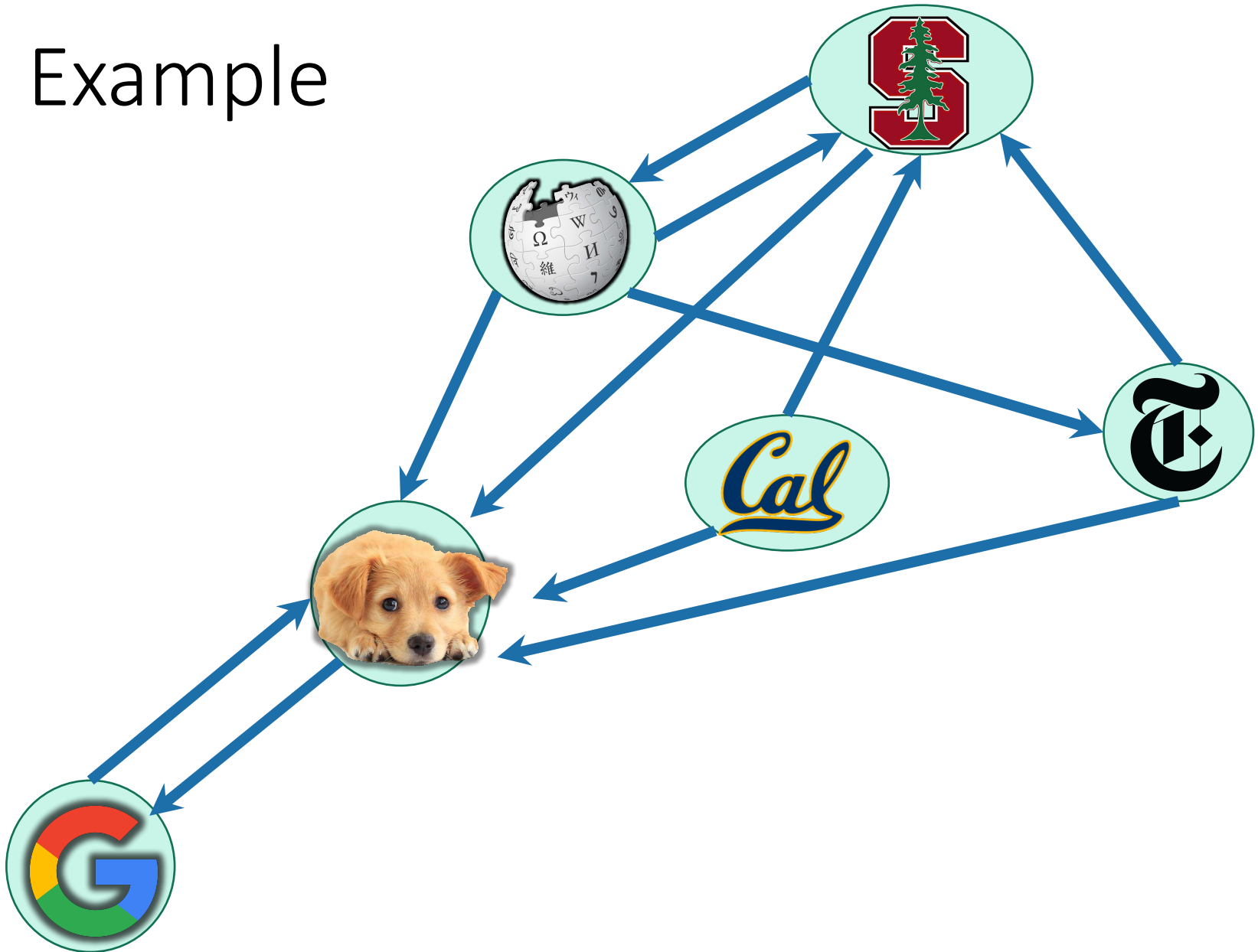
What if I told you I could do it in time $O(n+m)$?

Algorithm:

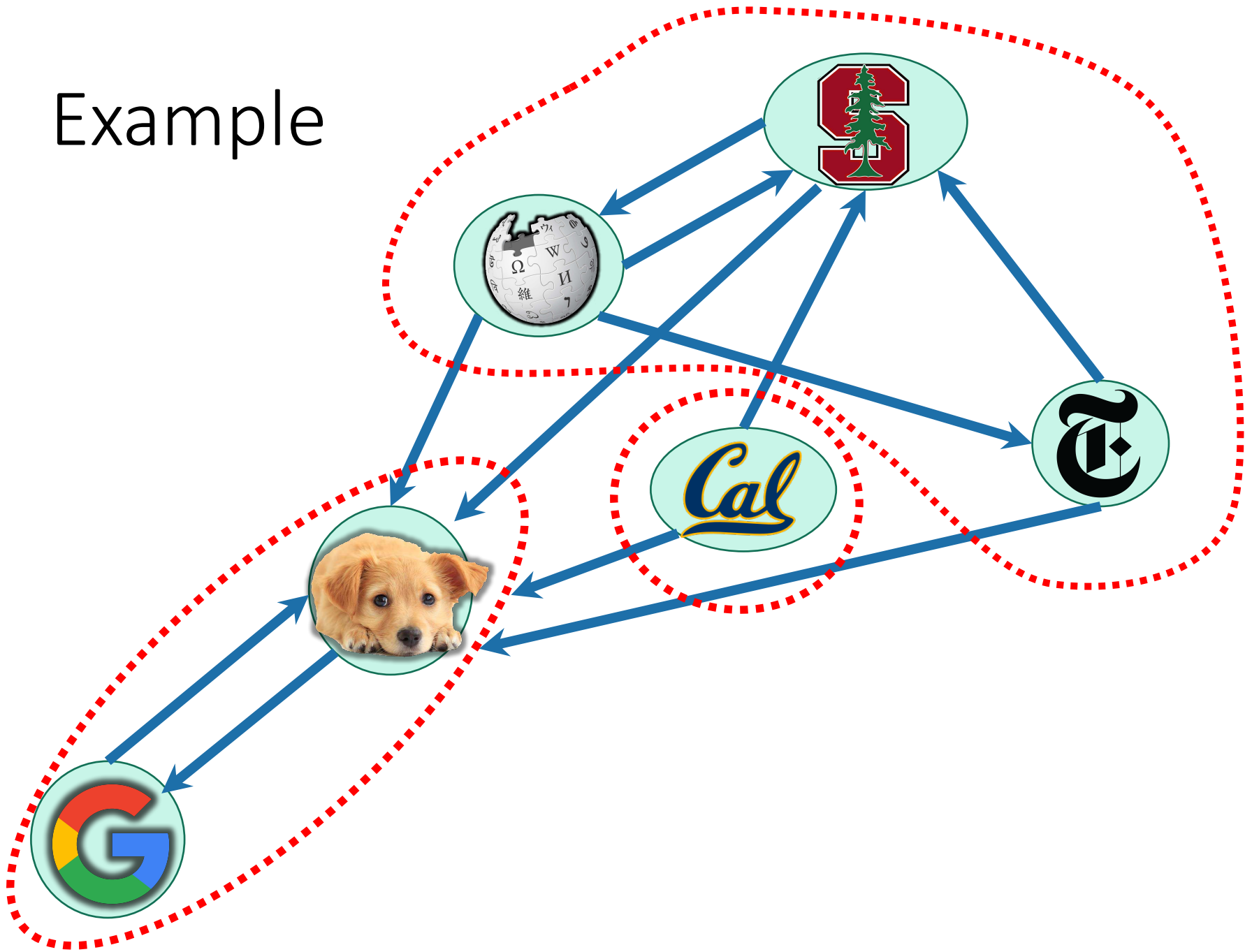
- Do DFS to create a **DFS forest**.
 - Choose starting vertices in any order.
 - Keep track of finishing times.
- **Reverse all the edges in the graph.**
- Do DFS again to create **another DFS forest**.
 - This time, order the nodes in the reverse order of the finishing times that they had from the first DFS run.
- The SCCs are the different trees in the **second DFS forest**.



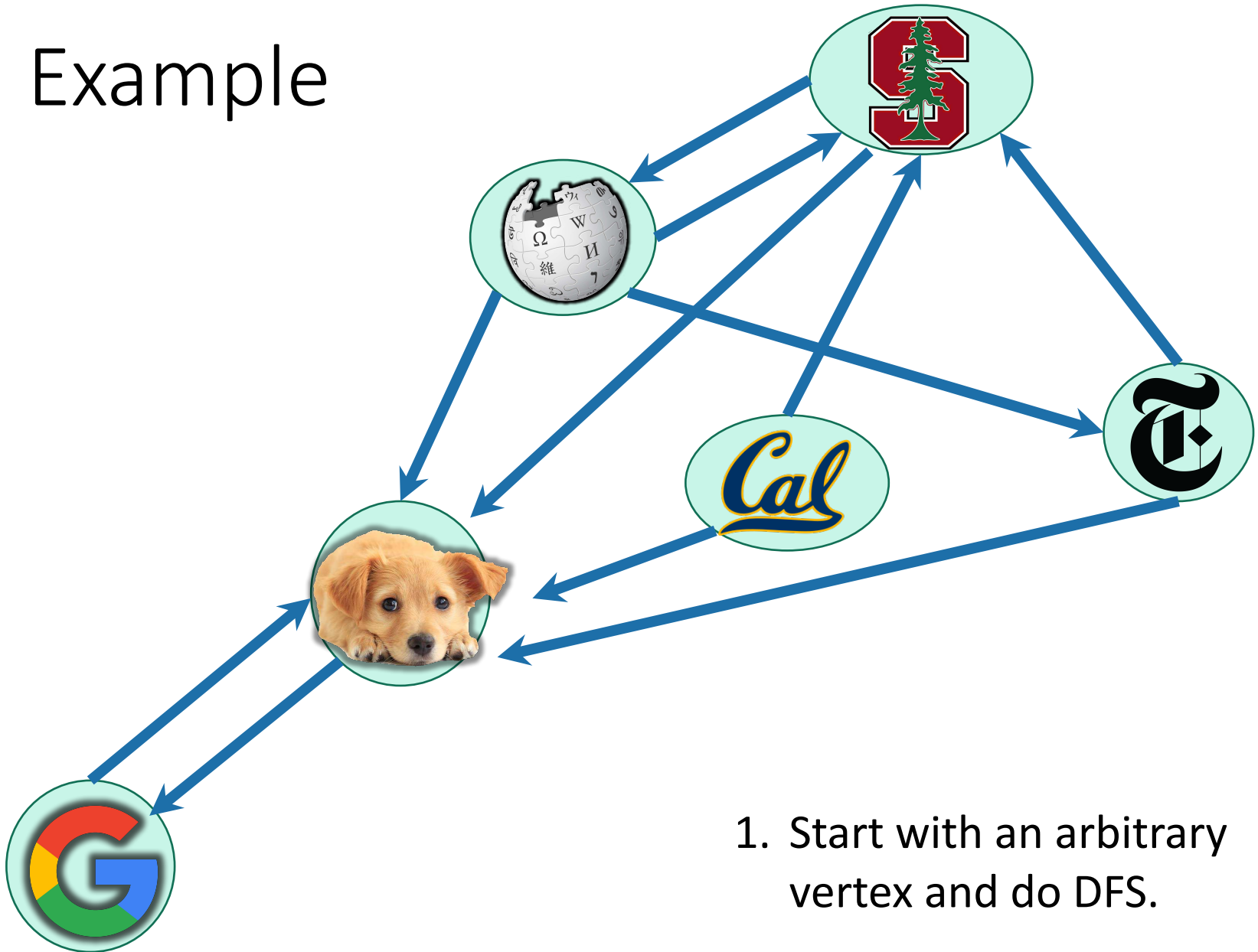
Example



Example

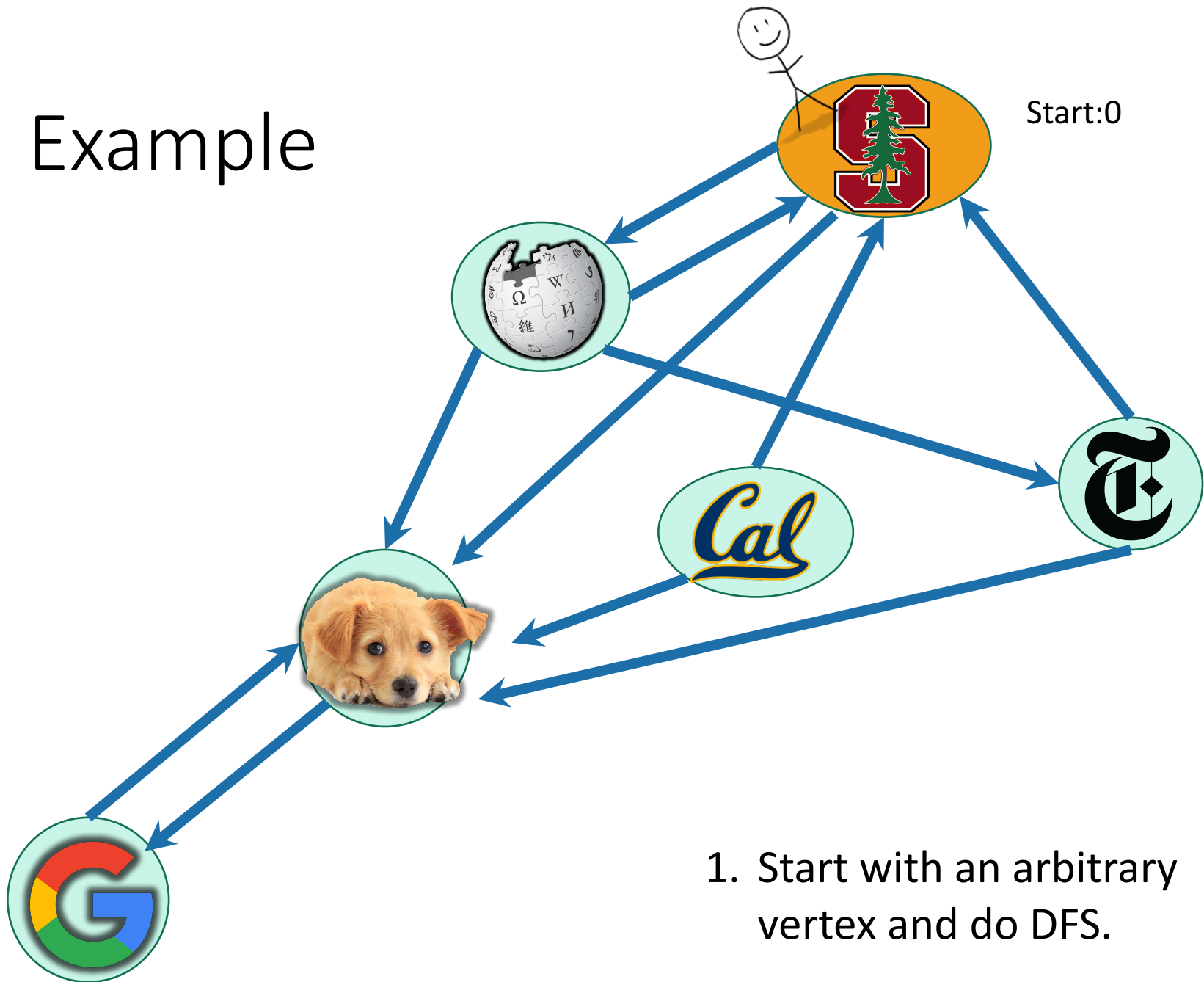


Example



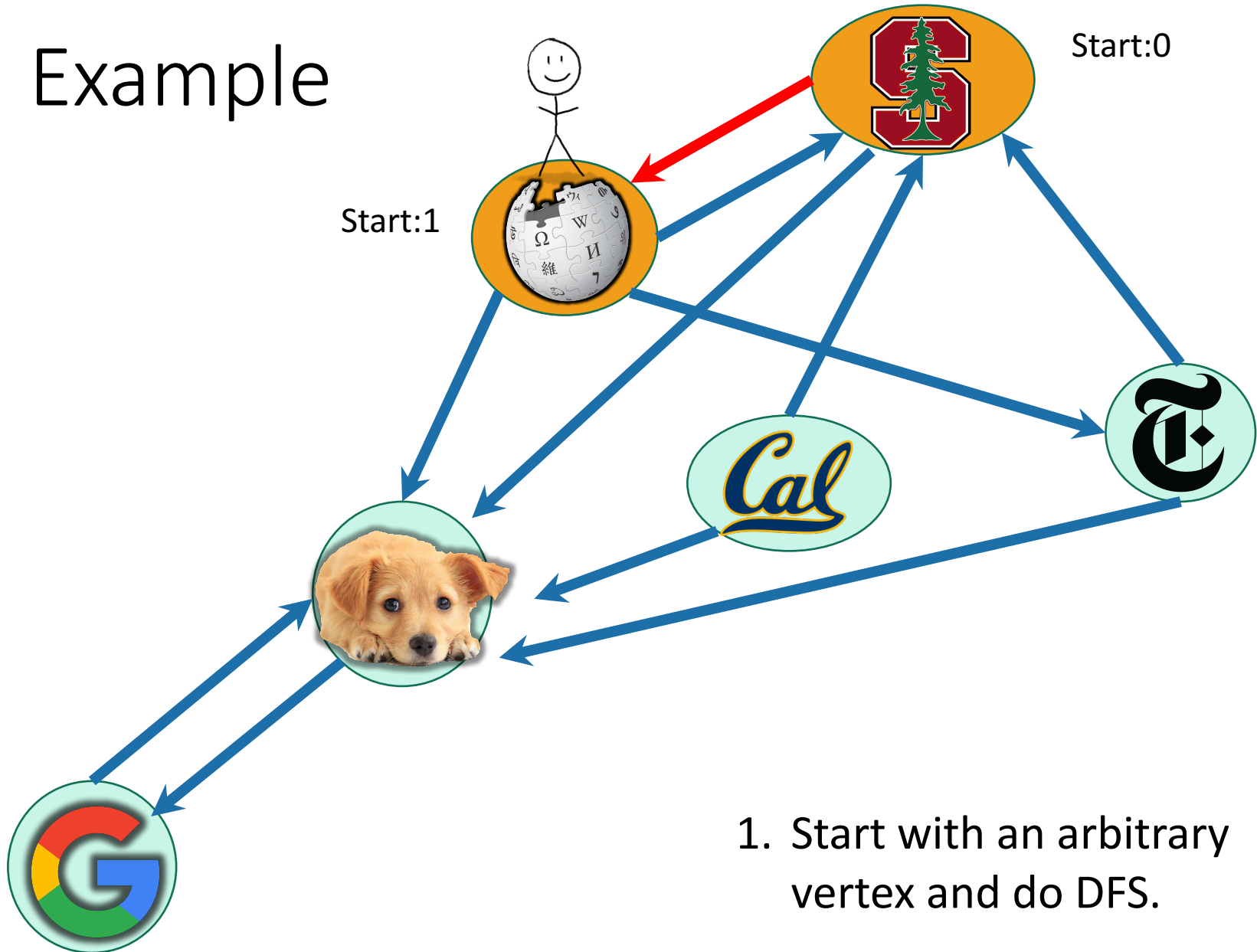
1. Start with an arbitrary vertex and do DFS.

Example



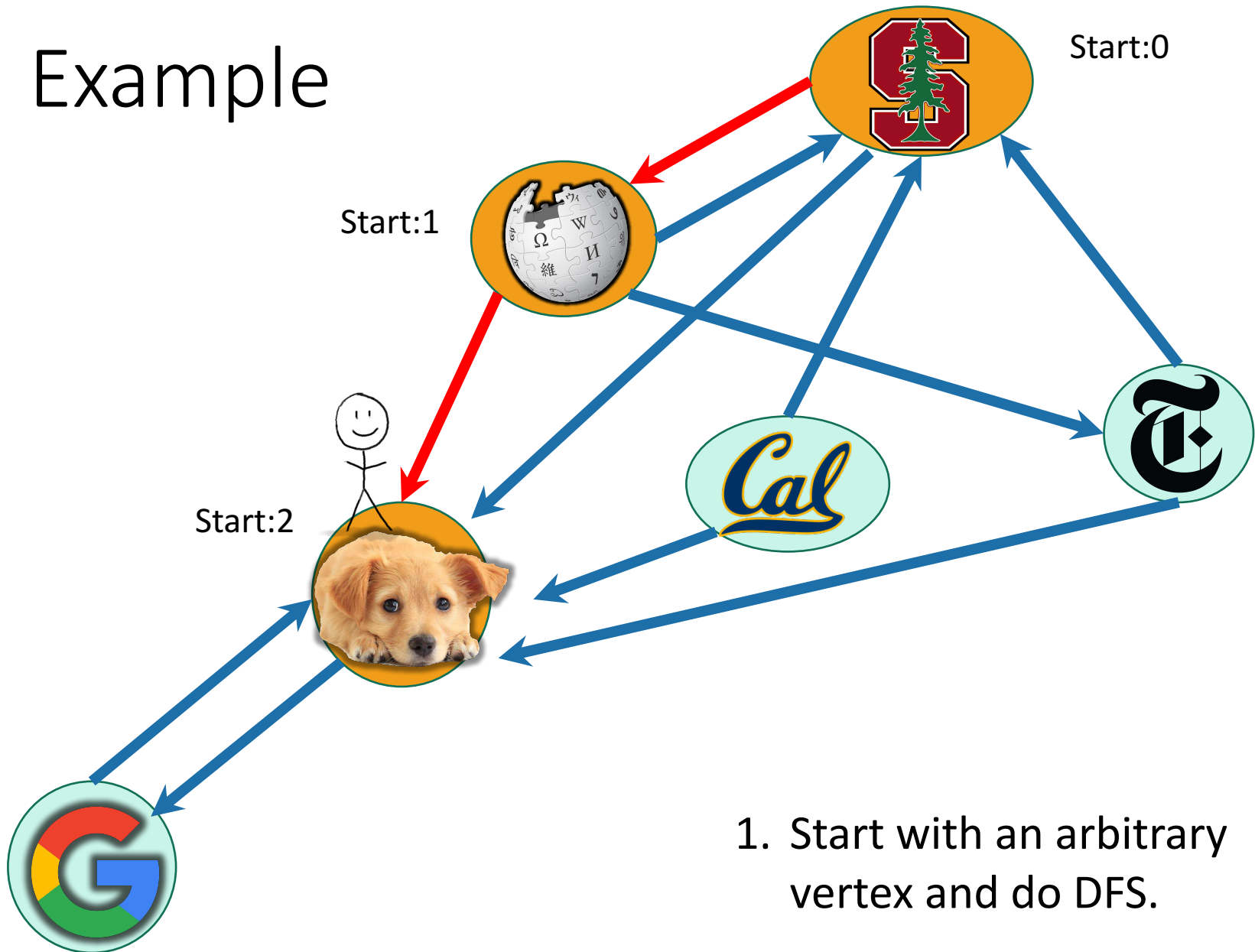
1. Start with an arbitrary vertex and do DFS.

Example



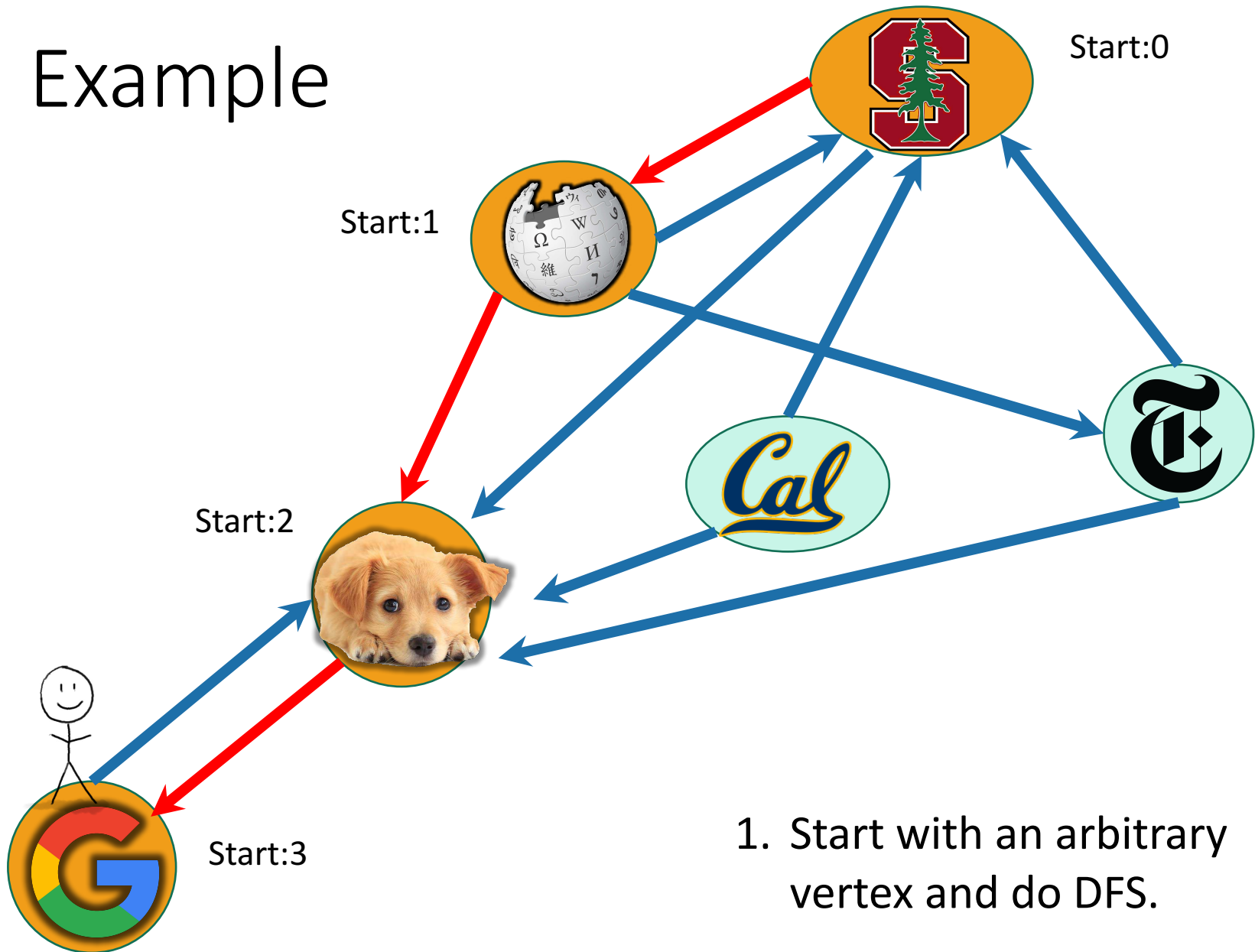
1. Start with an arbitrary vertex and do DFS.

Example



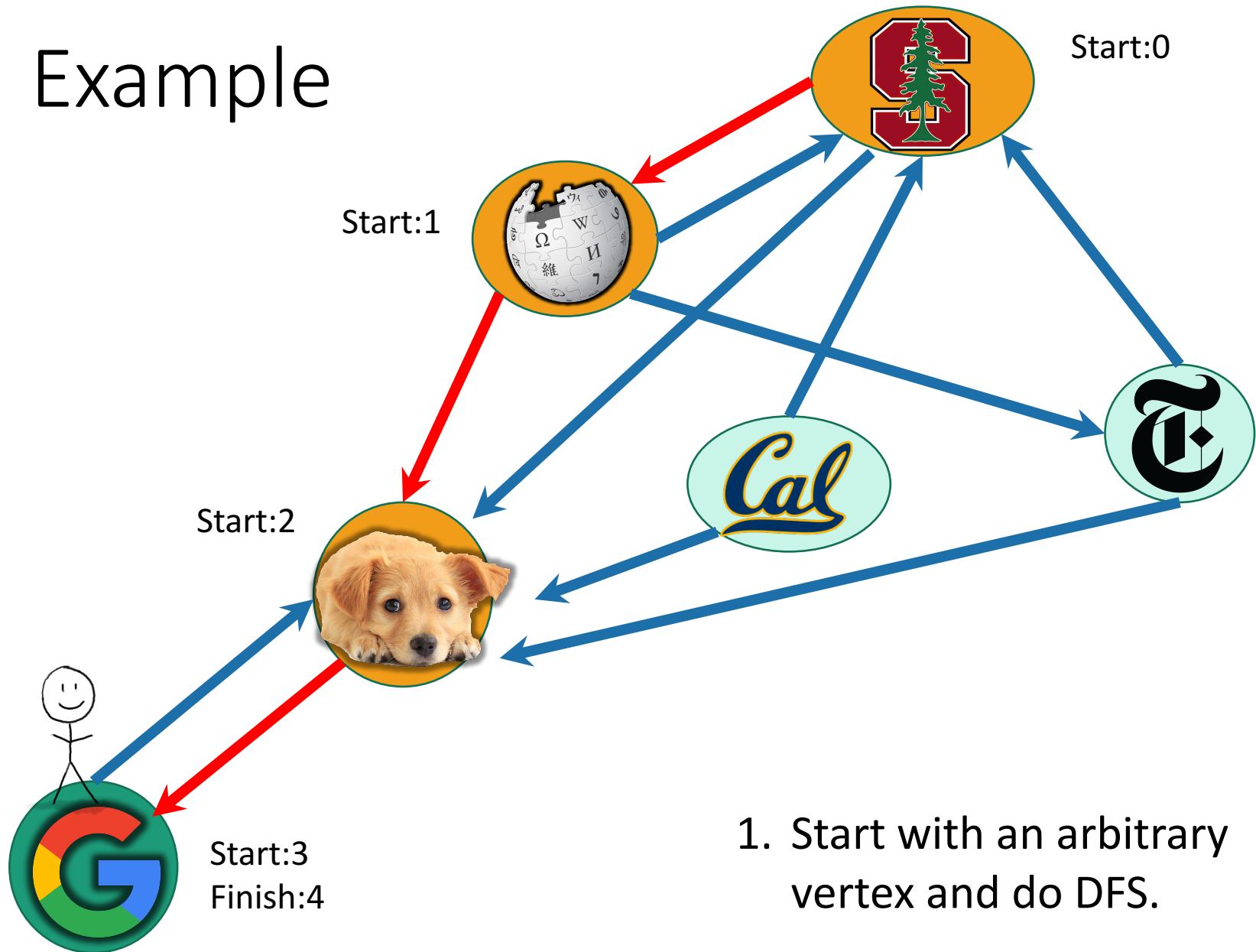
1. Start with an arbitrary vertex and do DFS.

Example



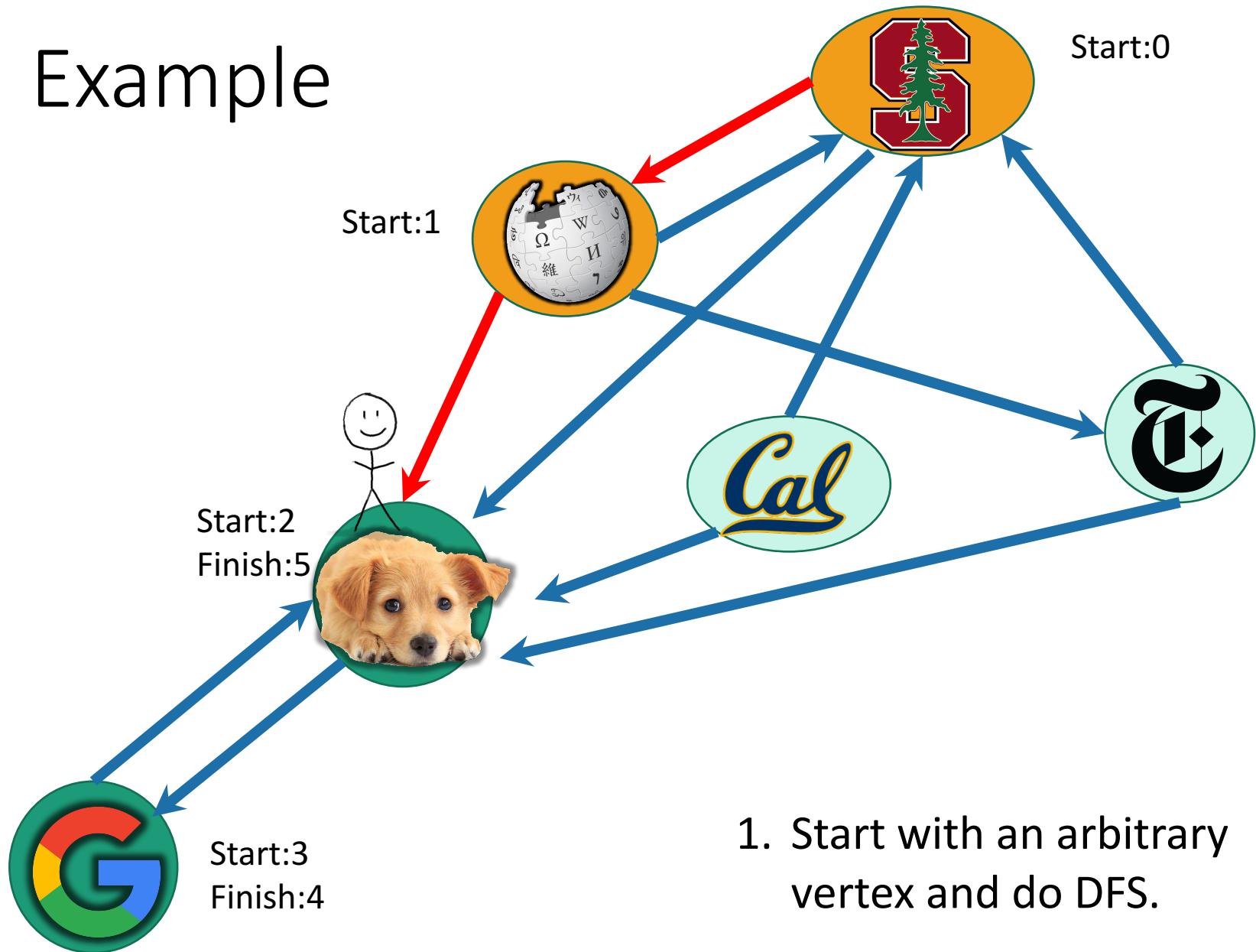
1. Start with an arbitrary vertex and do DFS.

Example



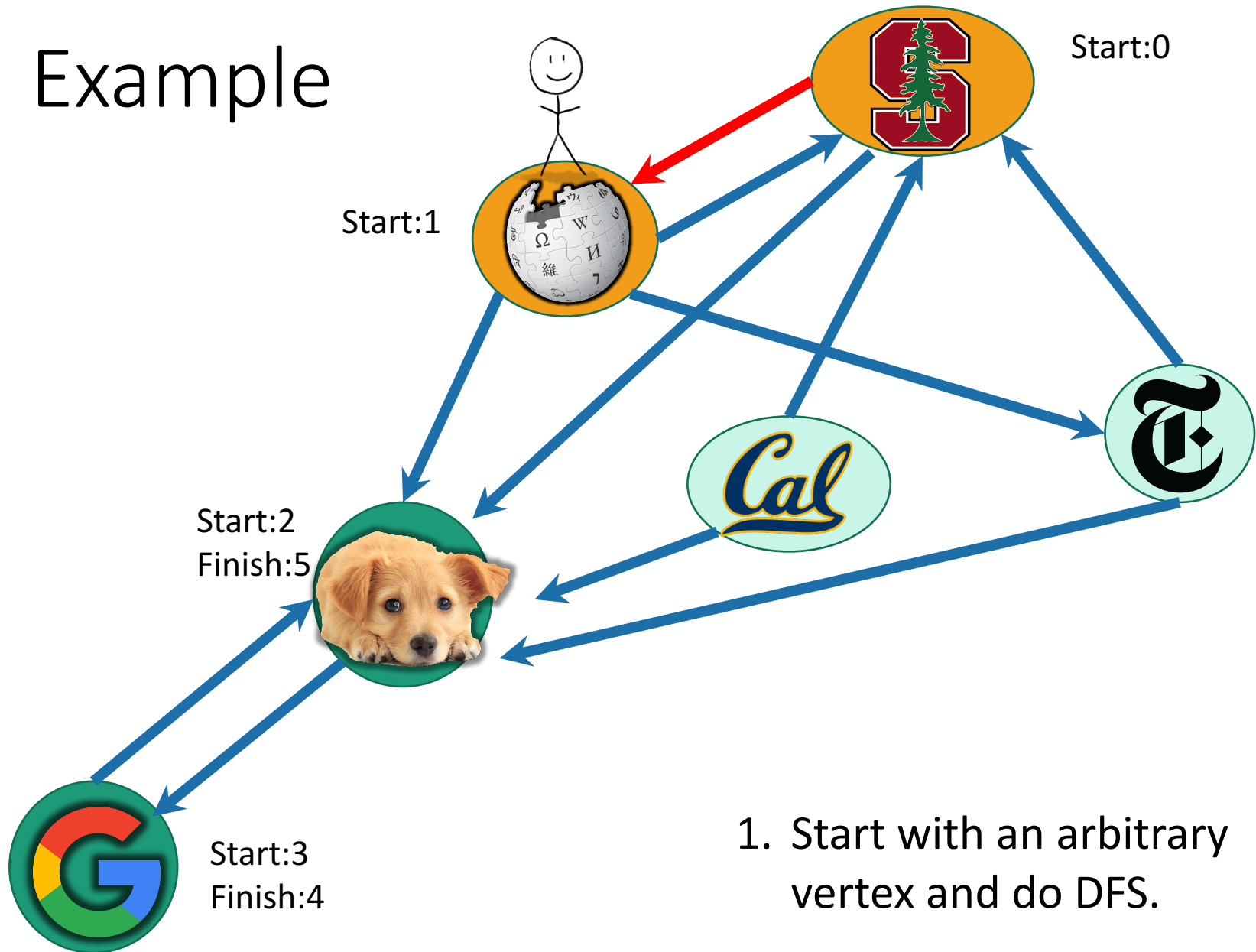
1. Start with an arbitrary vertex and do DFS.

Example

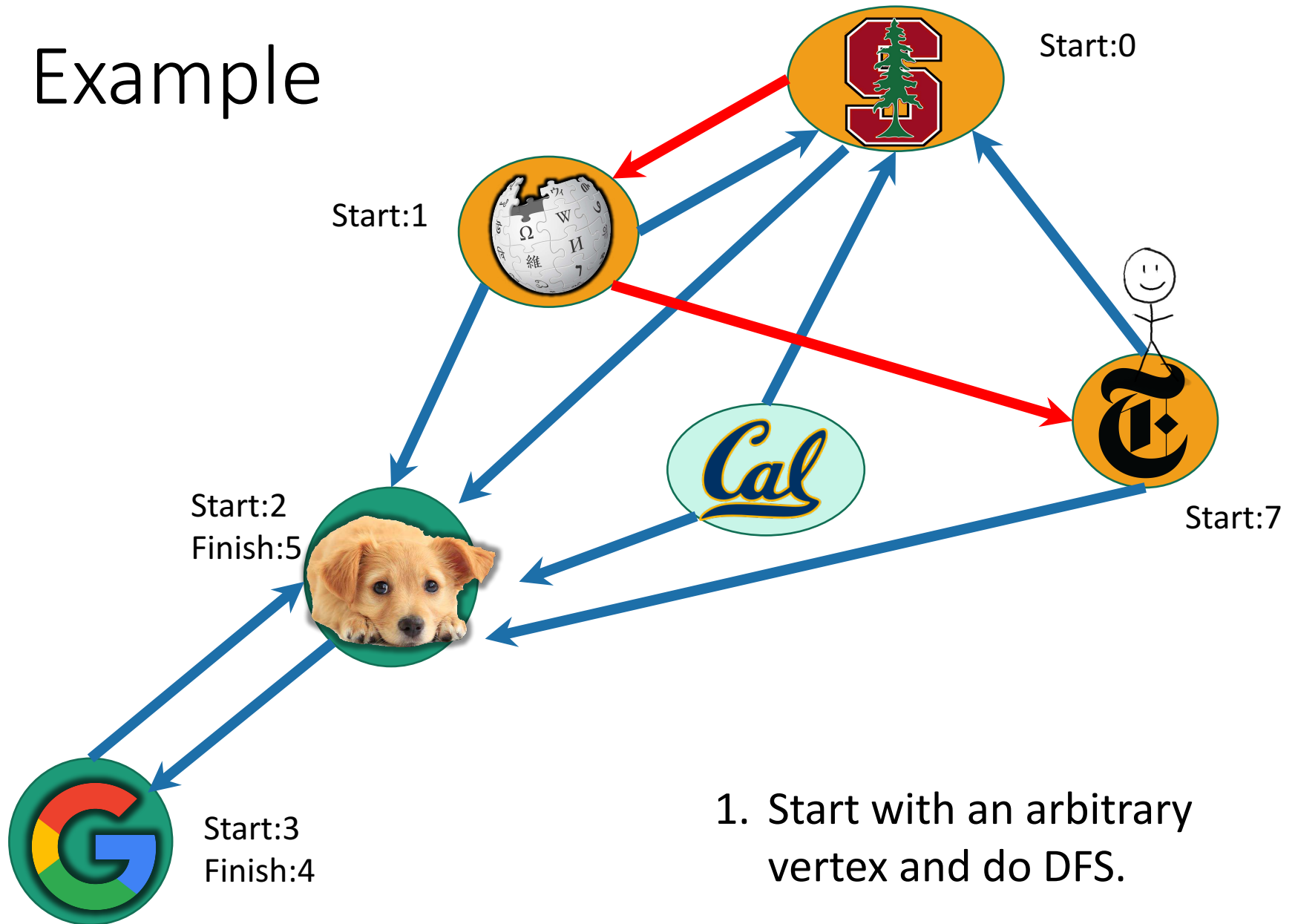


1. Start with an arbitrary vertex and do DFS.

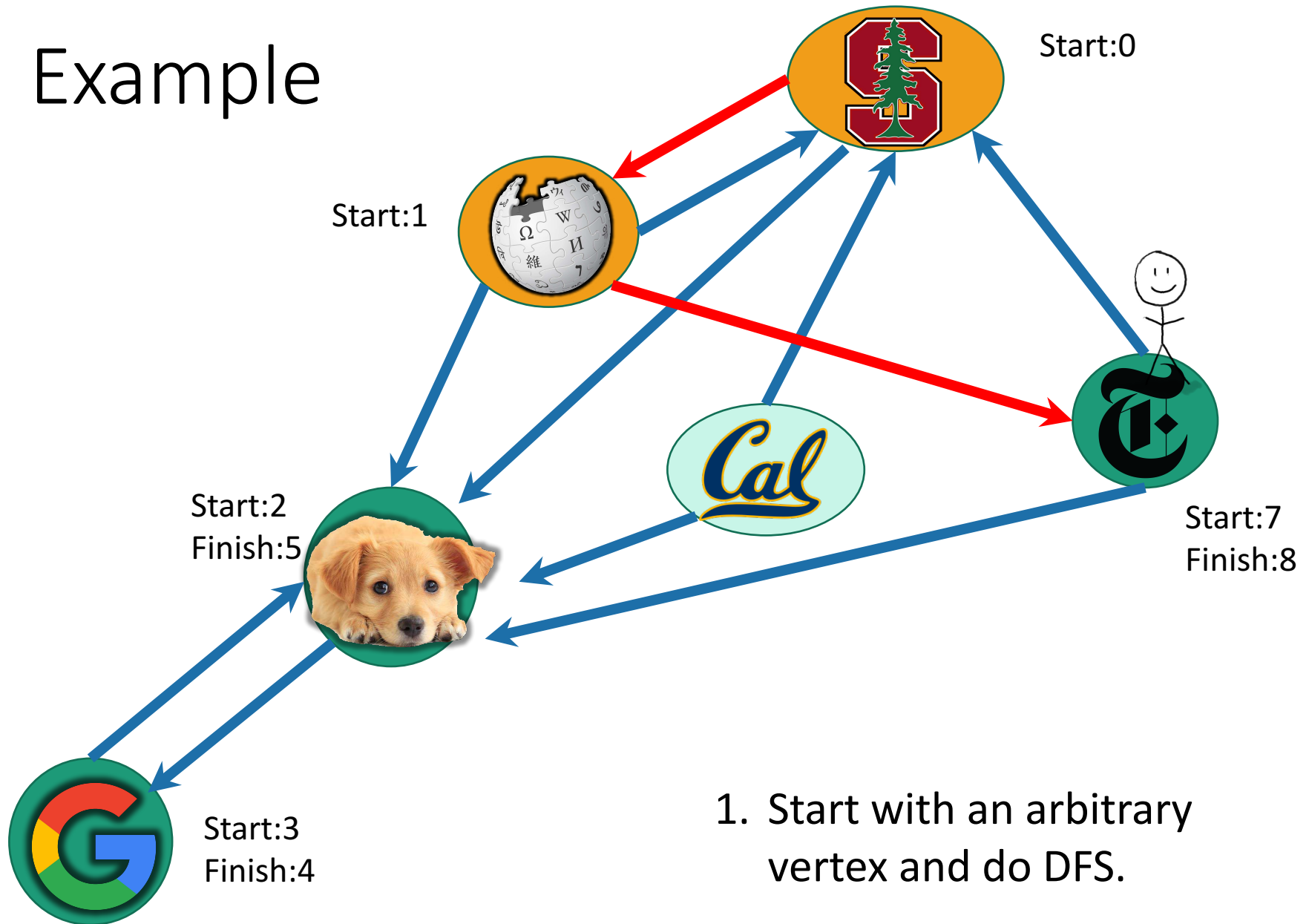
Example



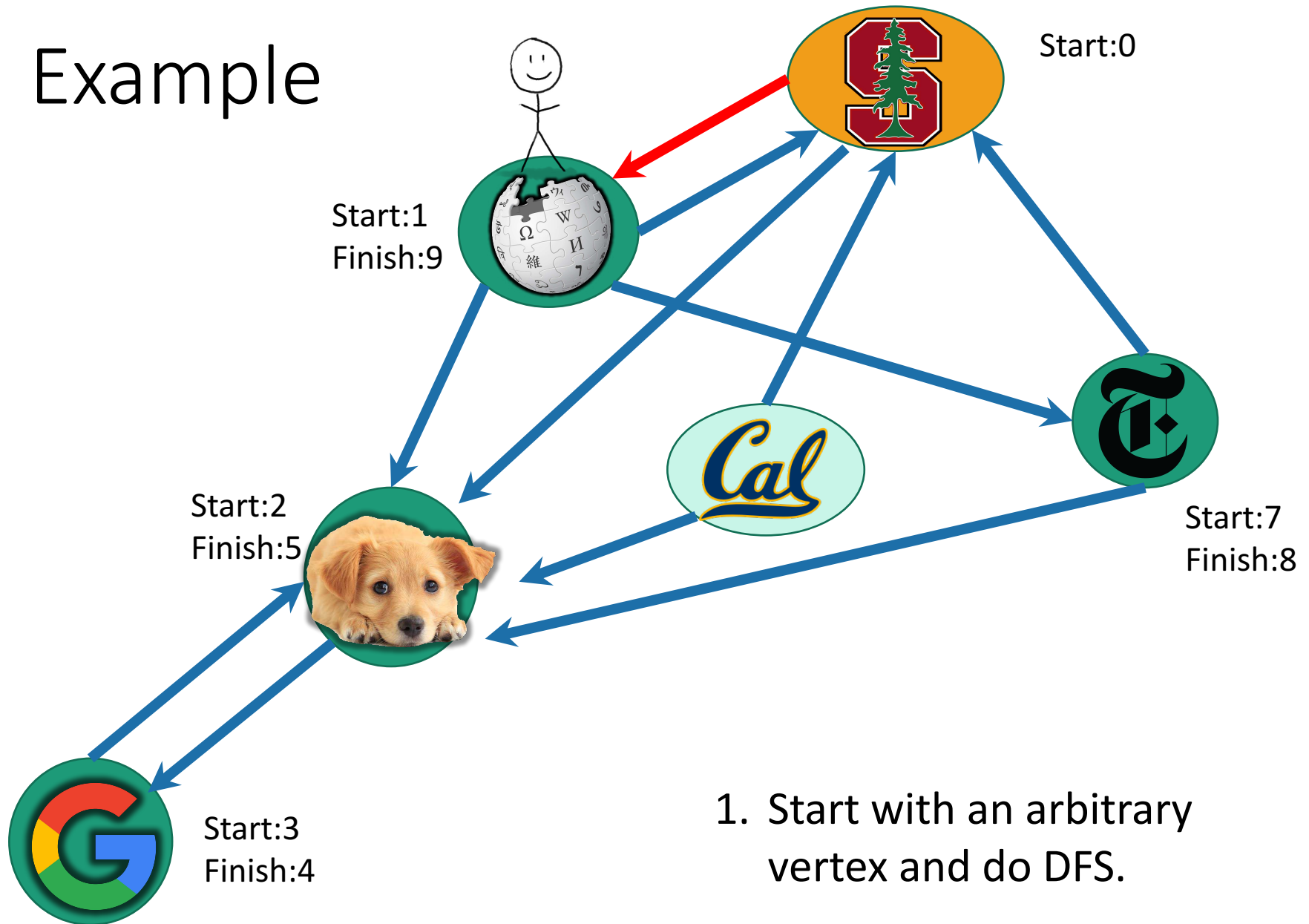
Example



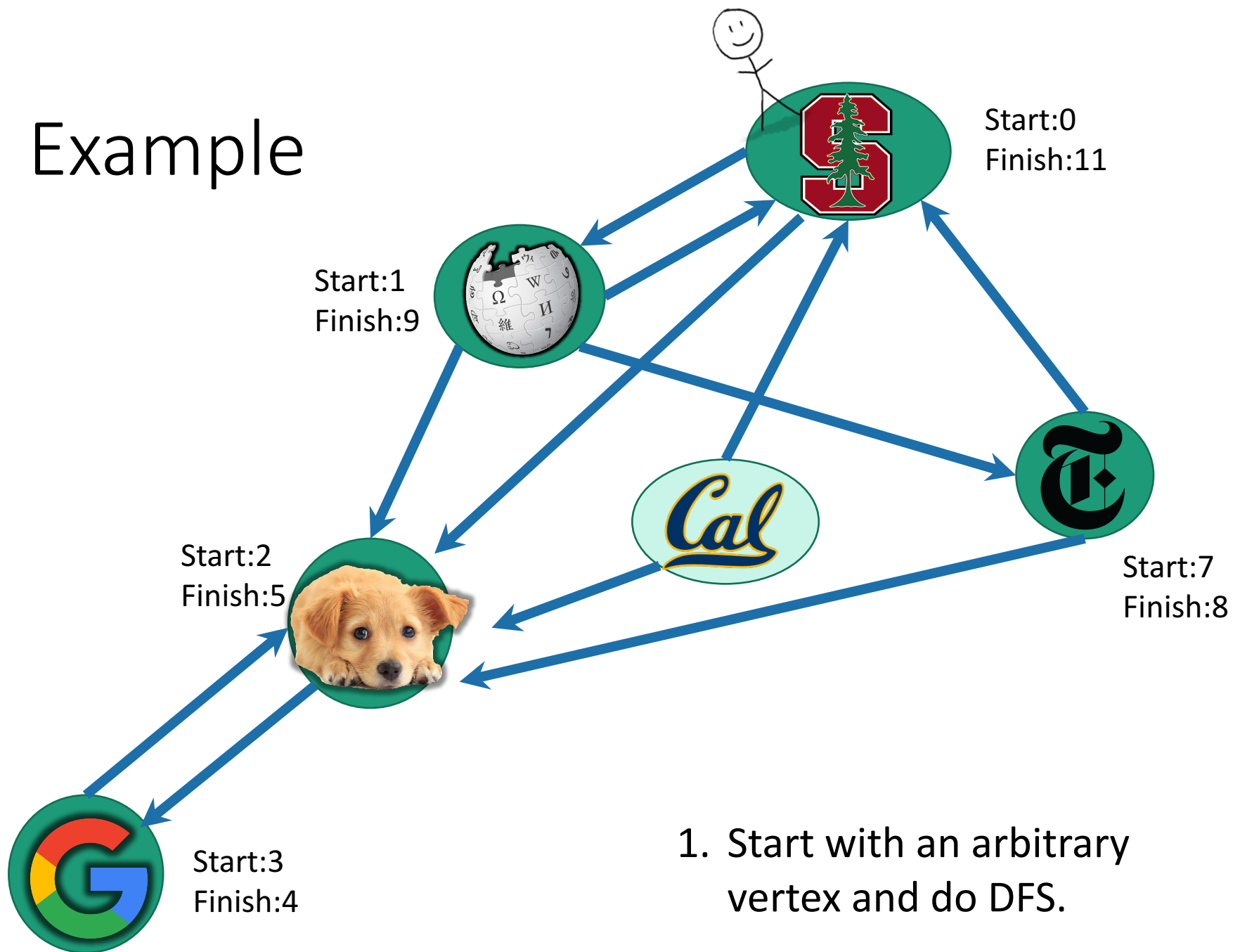
Example



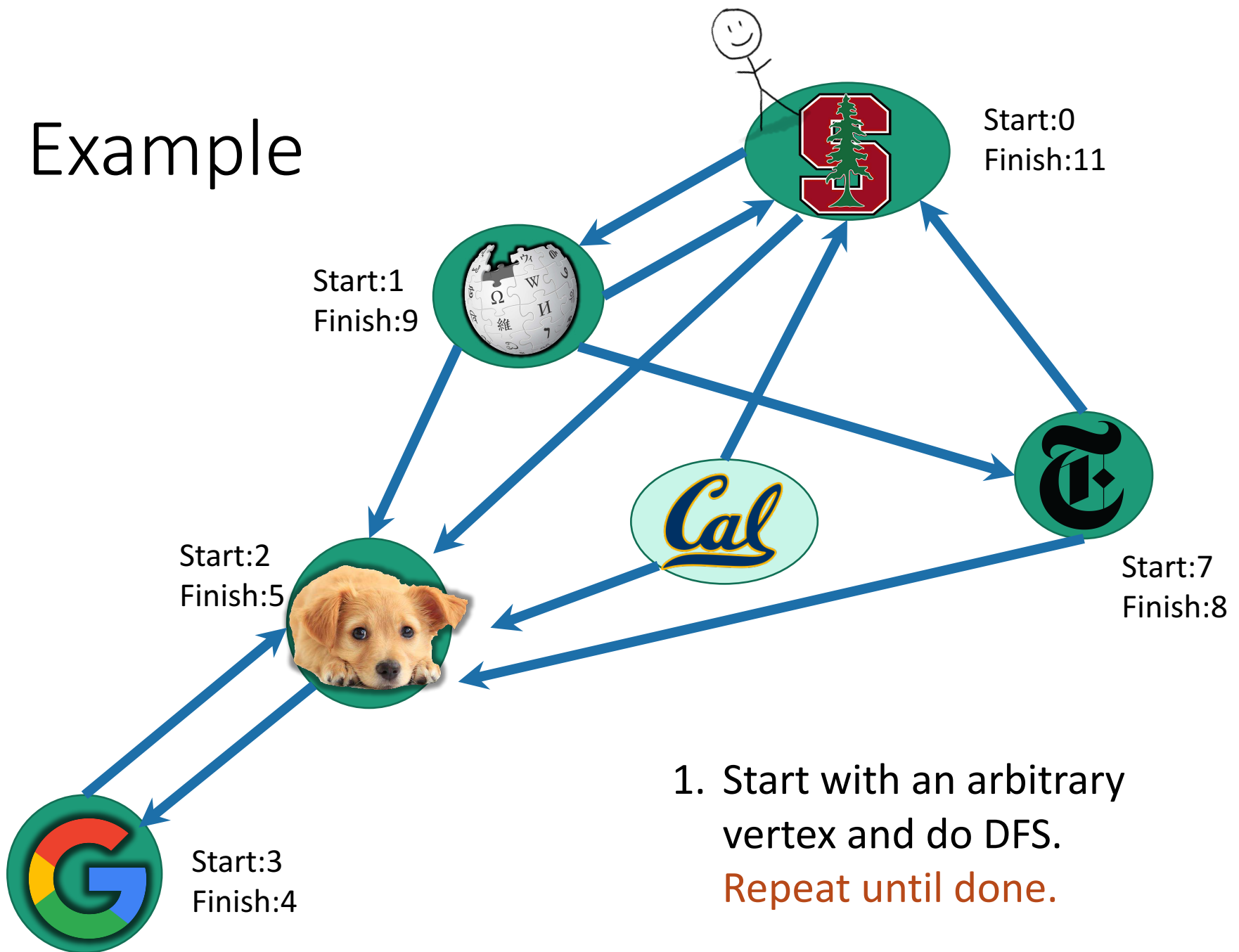
Example



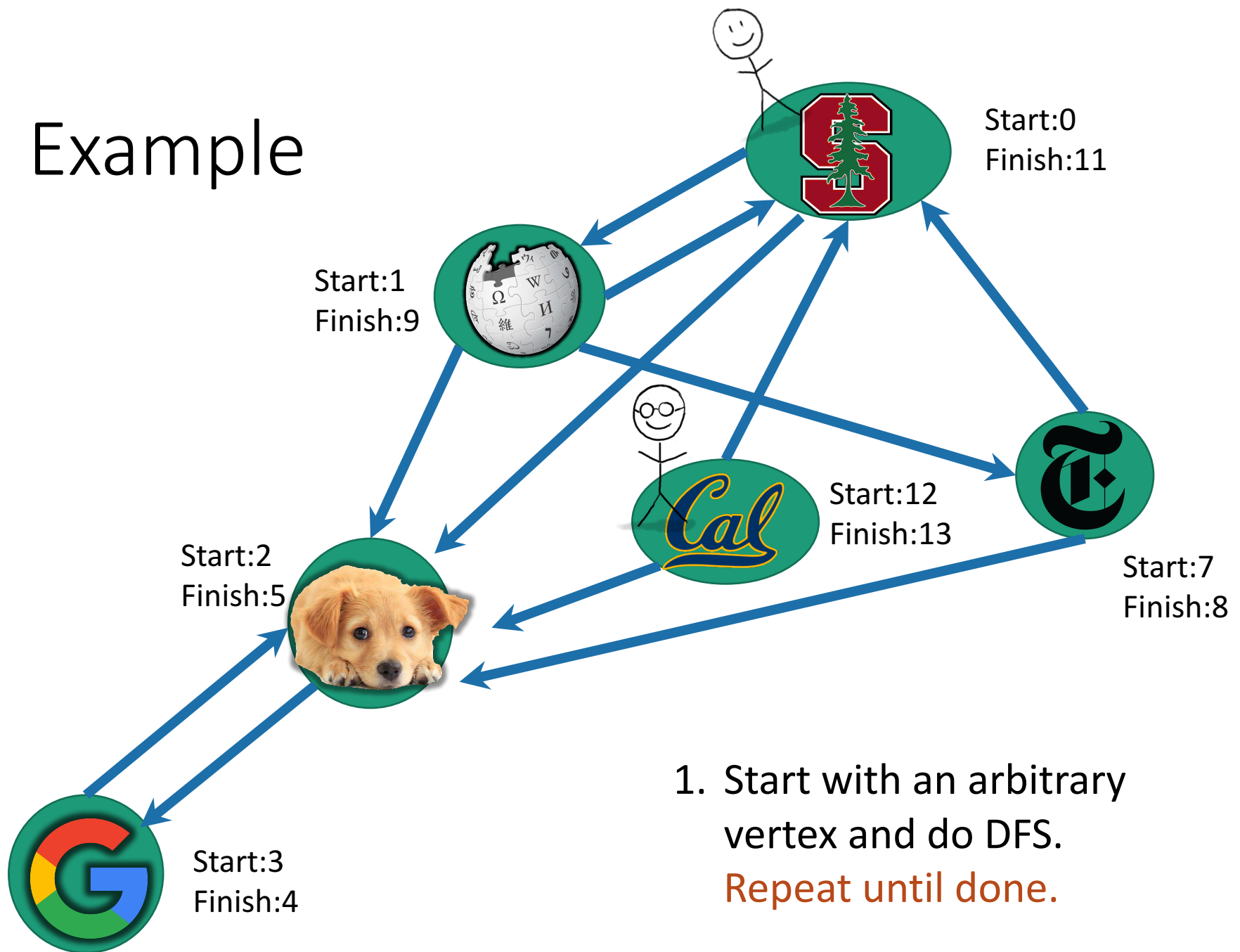
Example



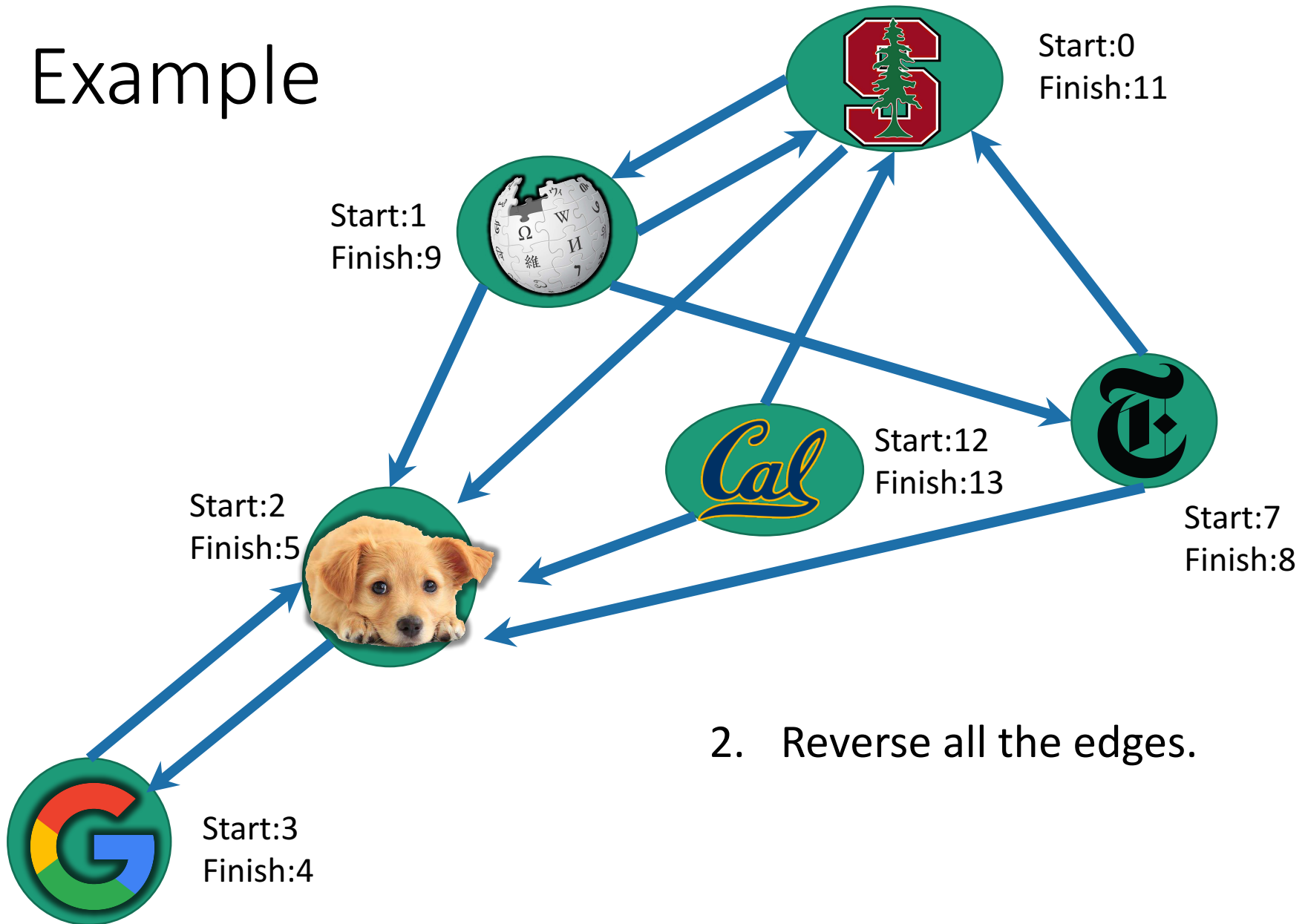
Example



Example

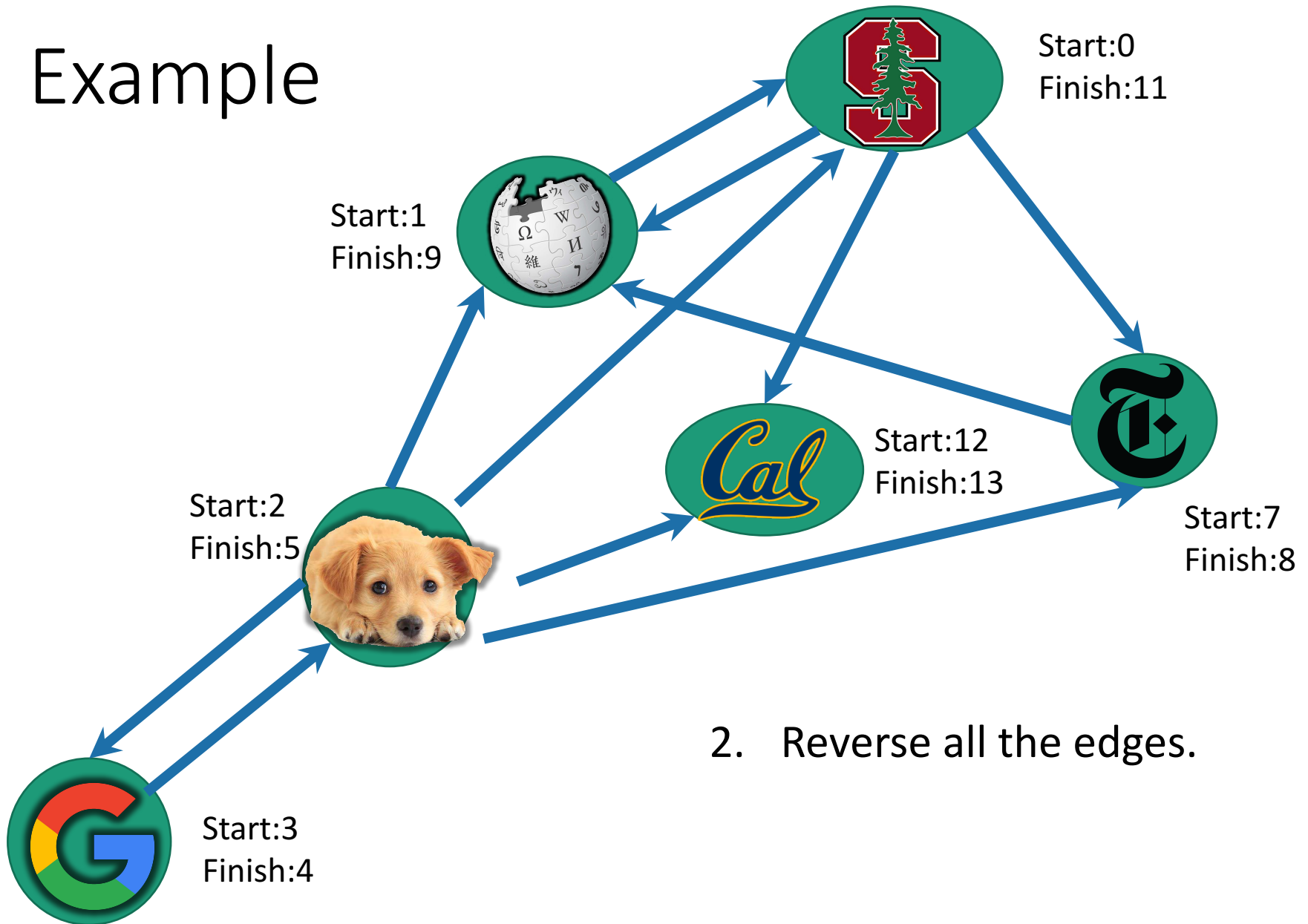


Example

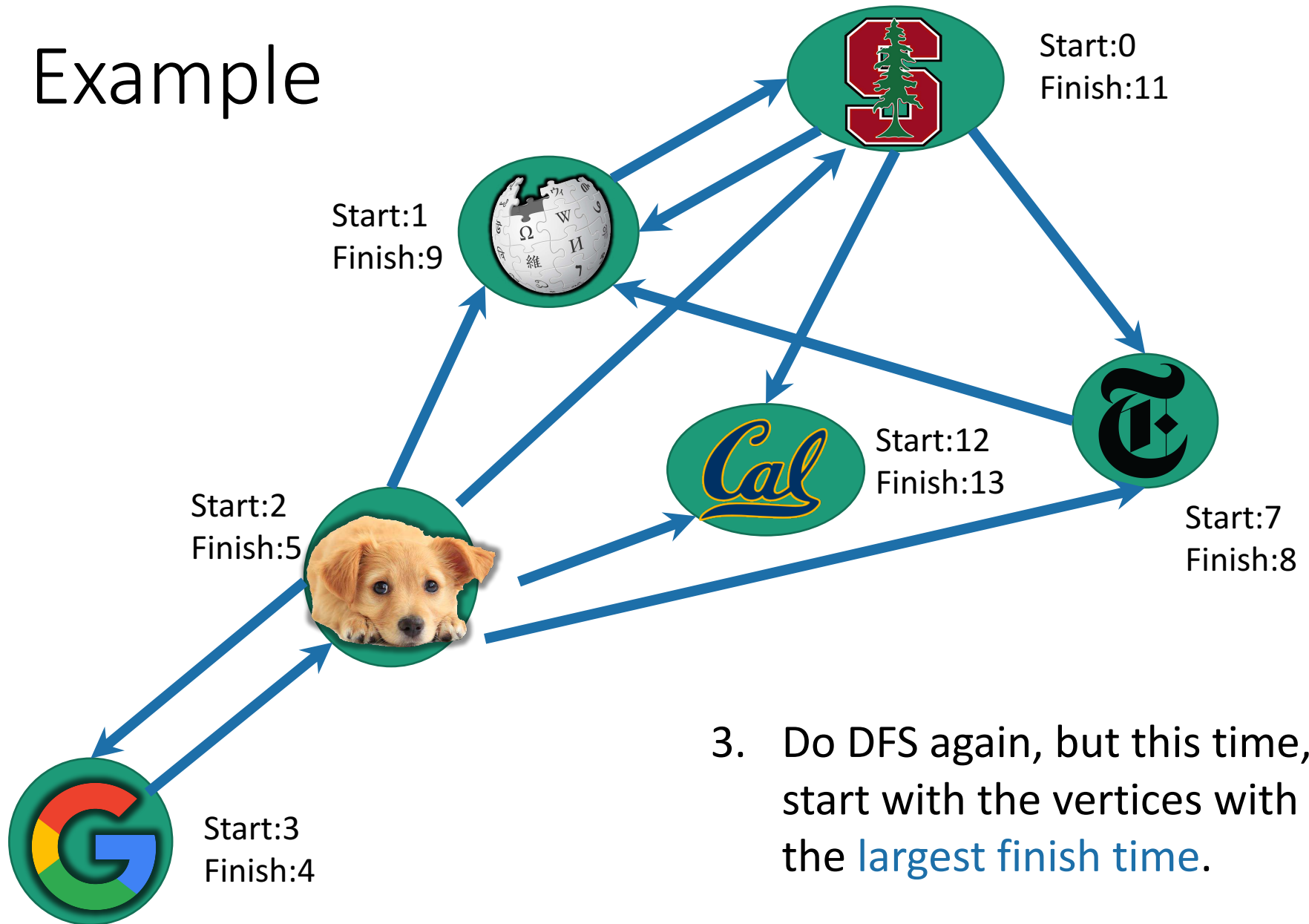


2. Reverse all the edges.

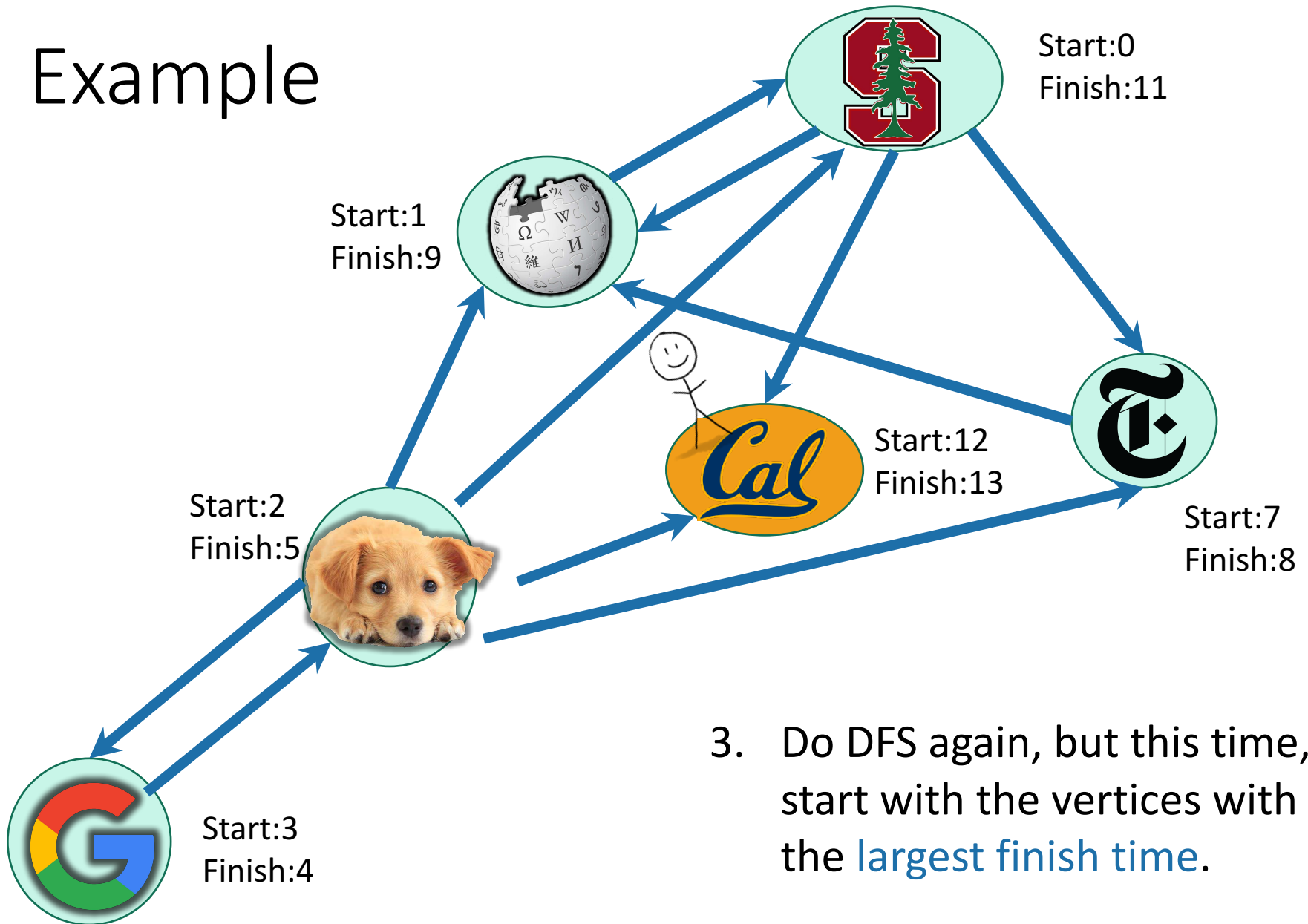
Example



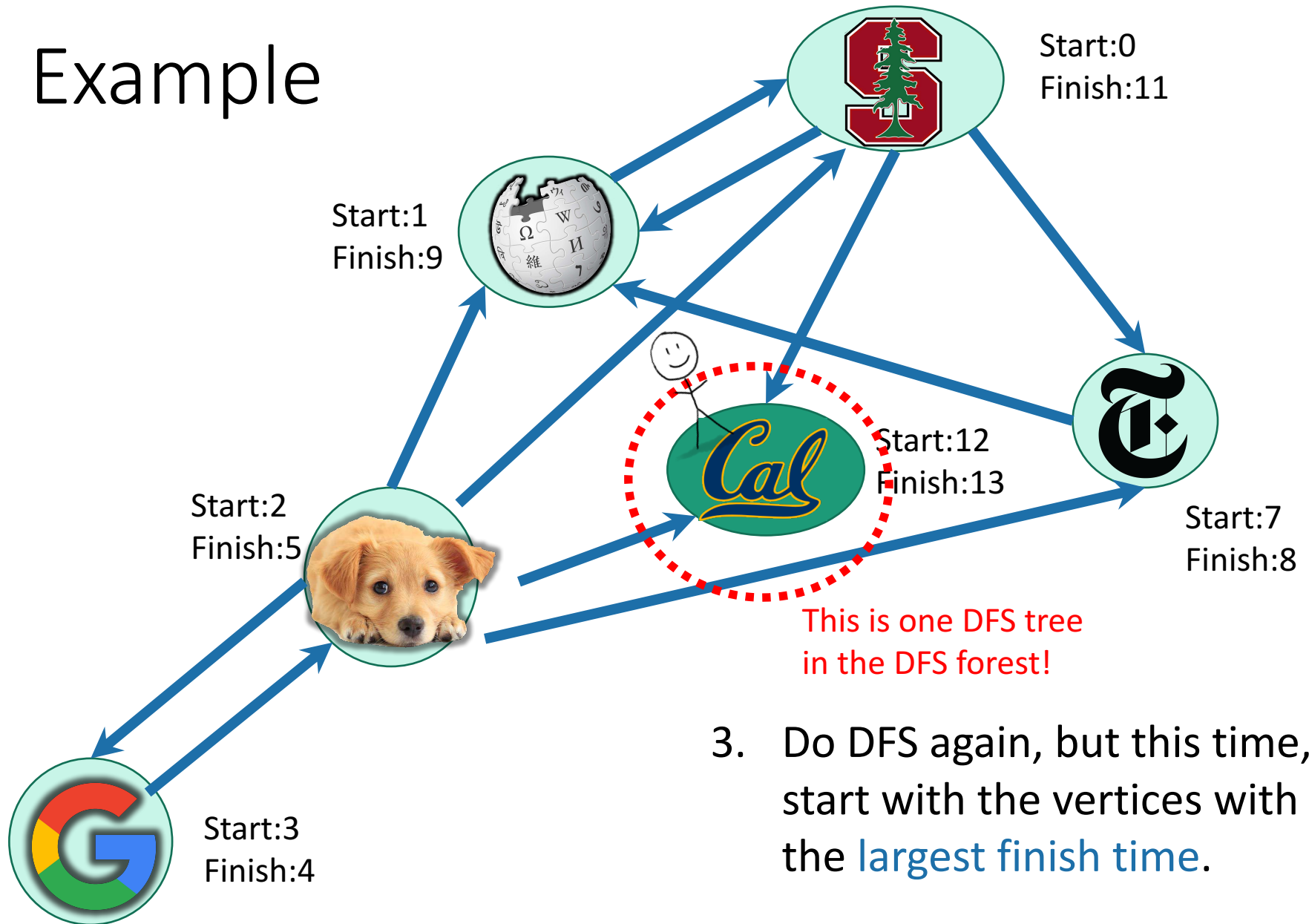
Example



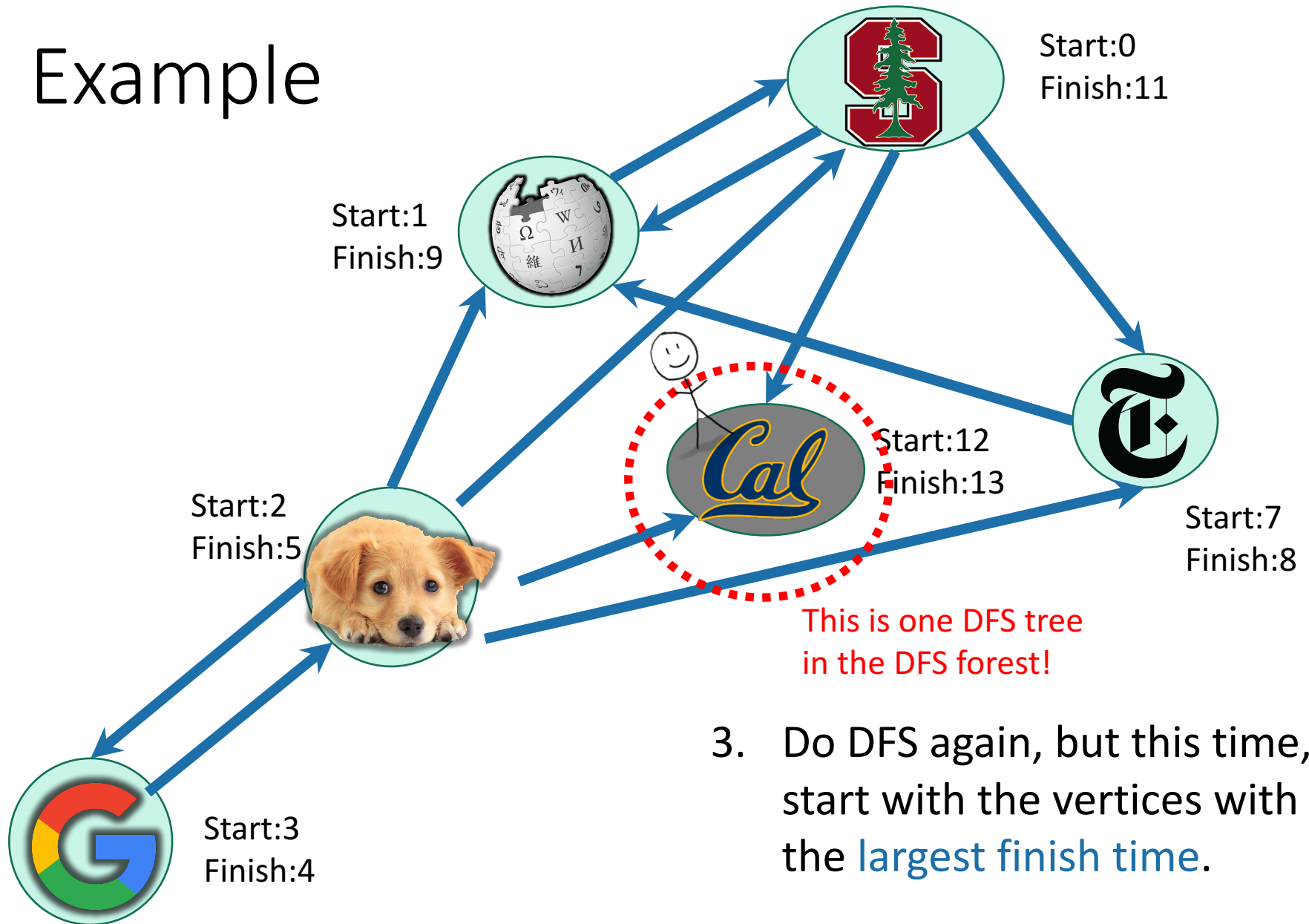
Example



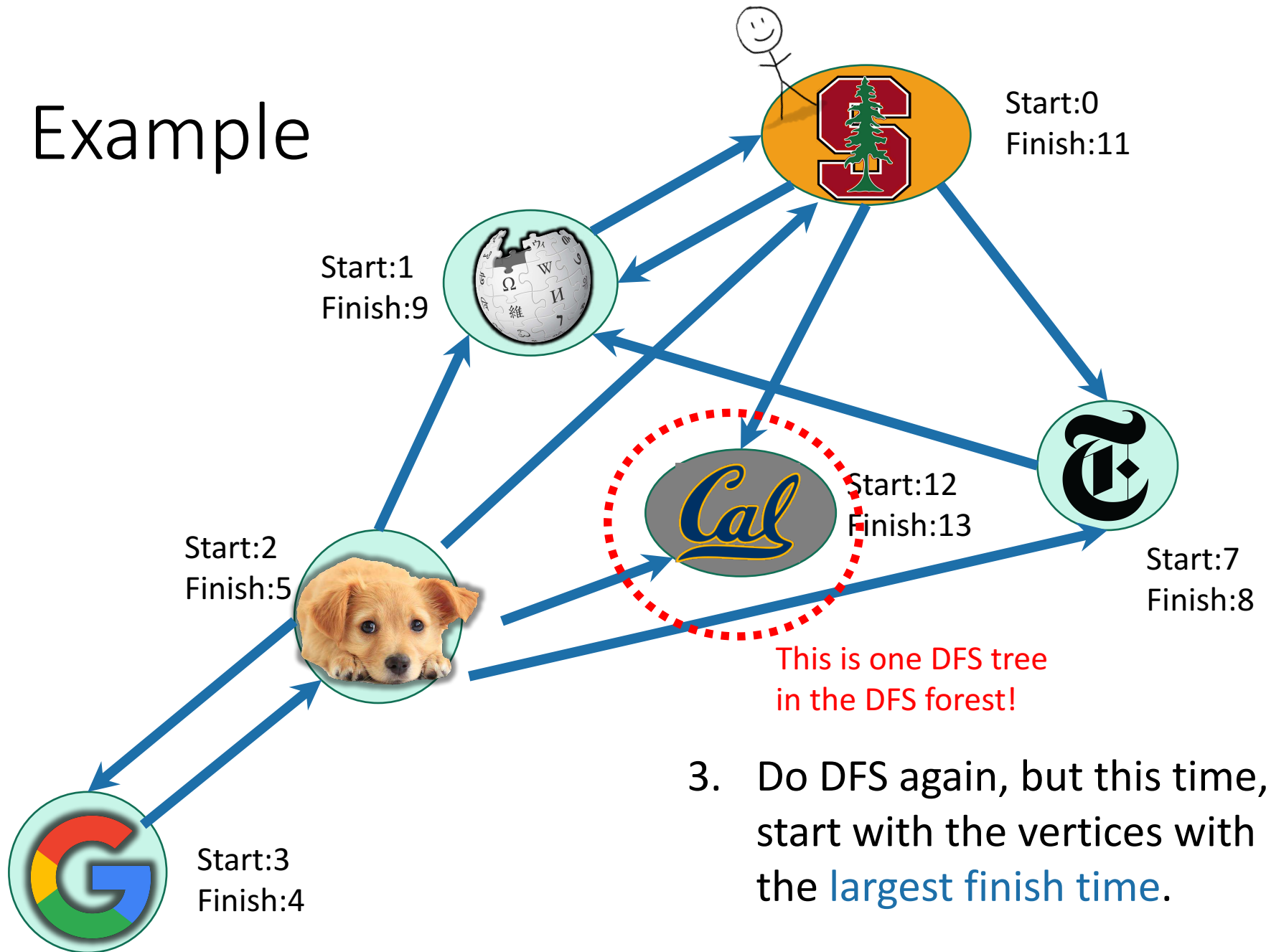
Example



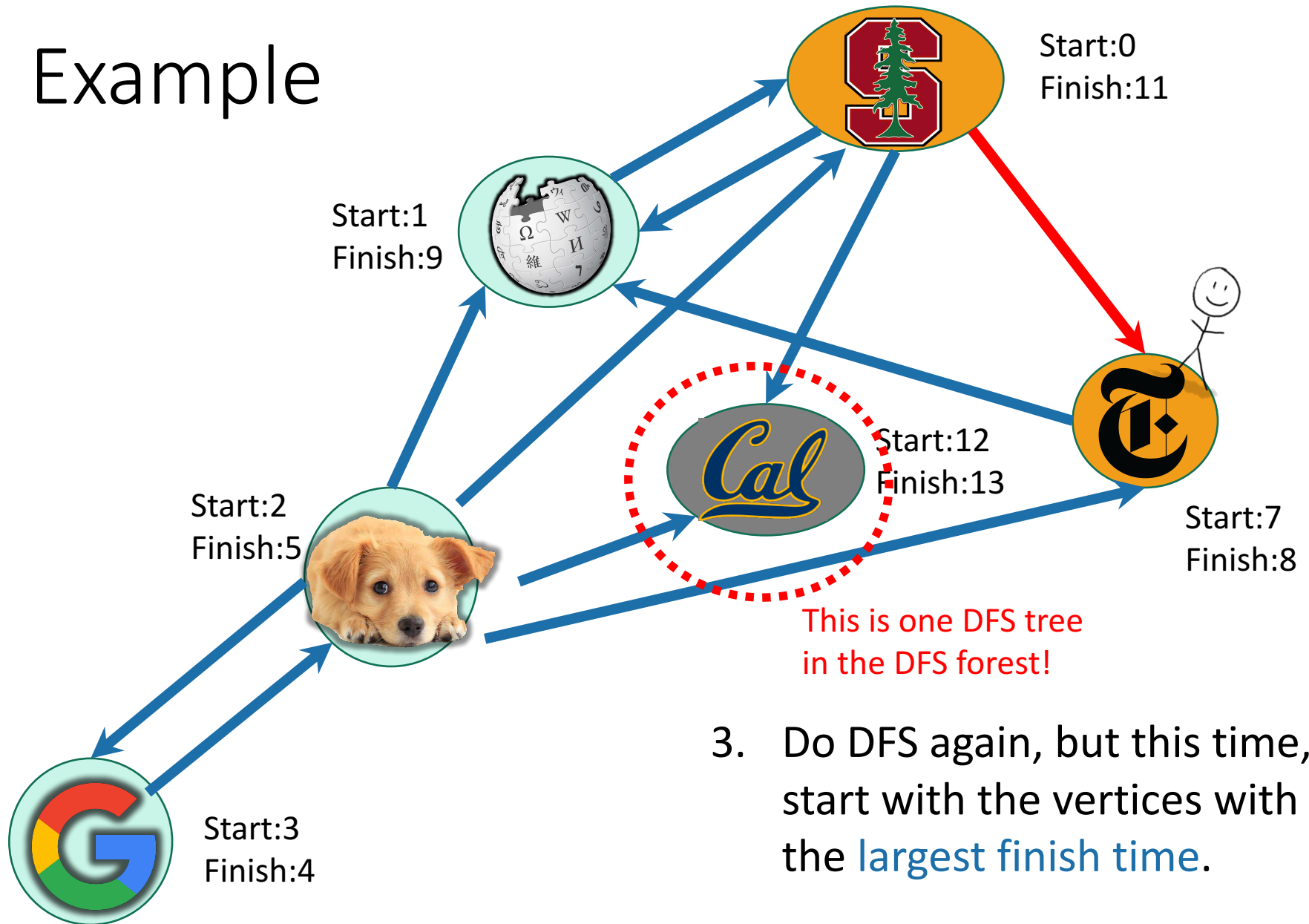
Example



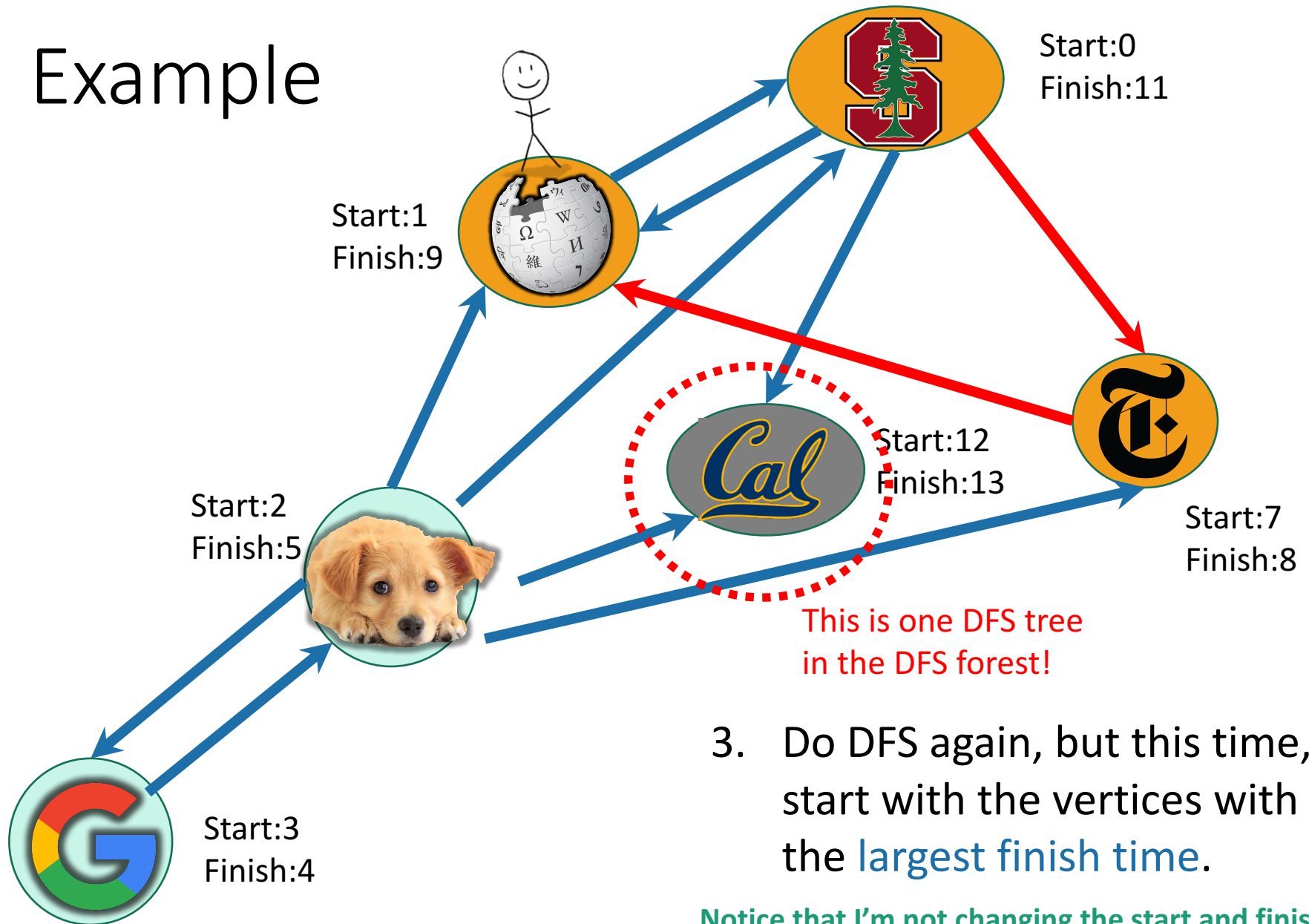
Example



Example

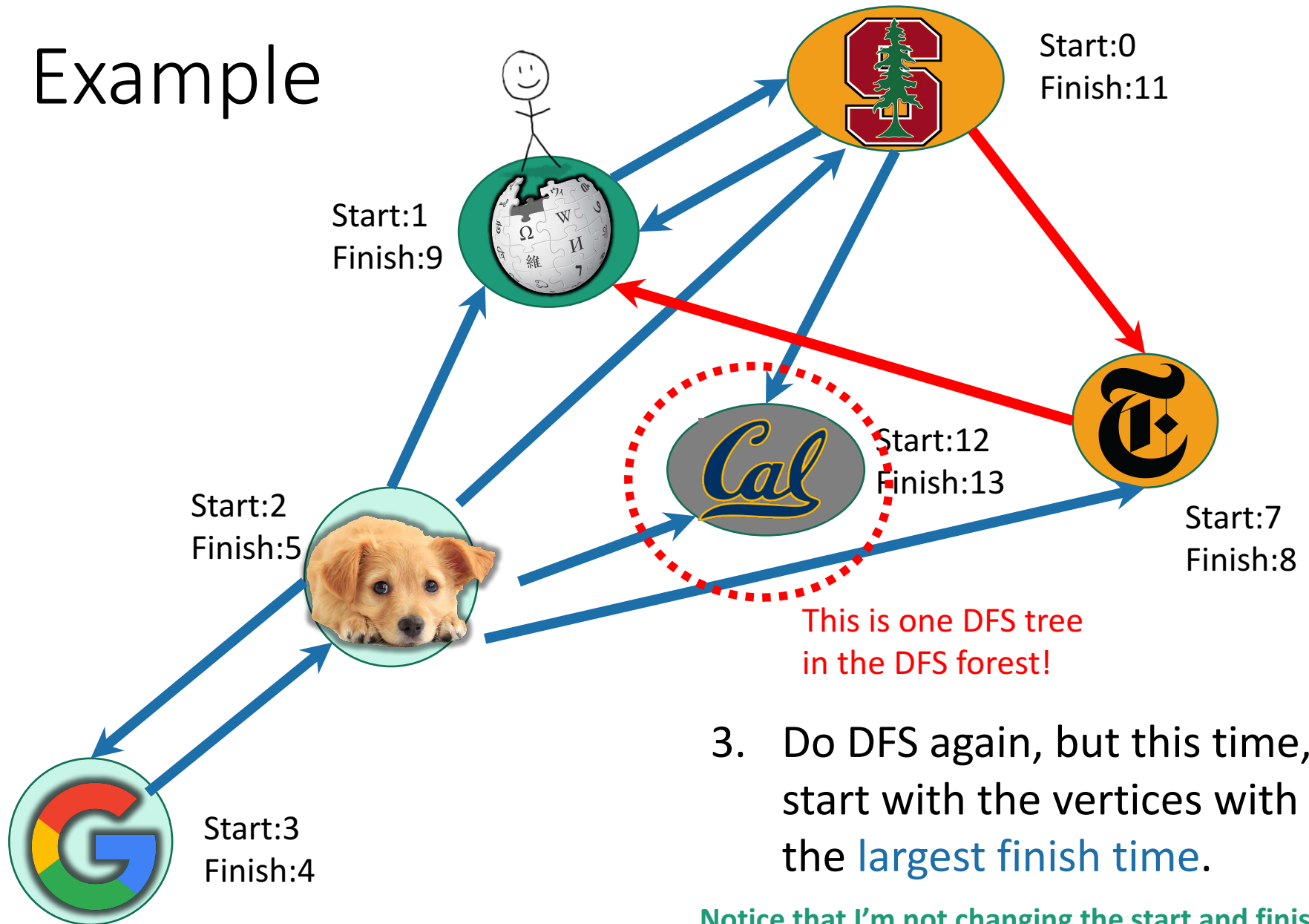


Example



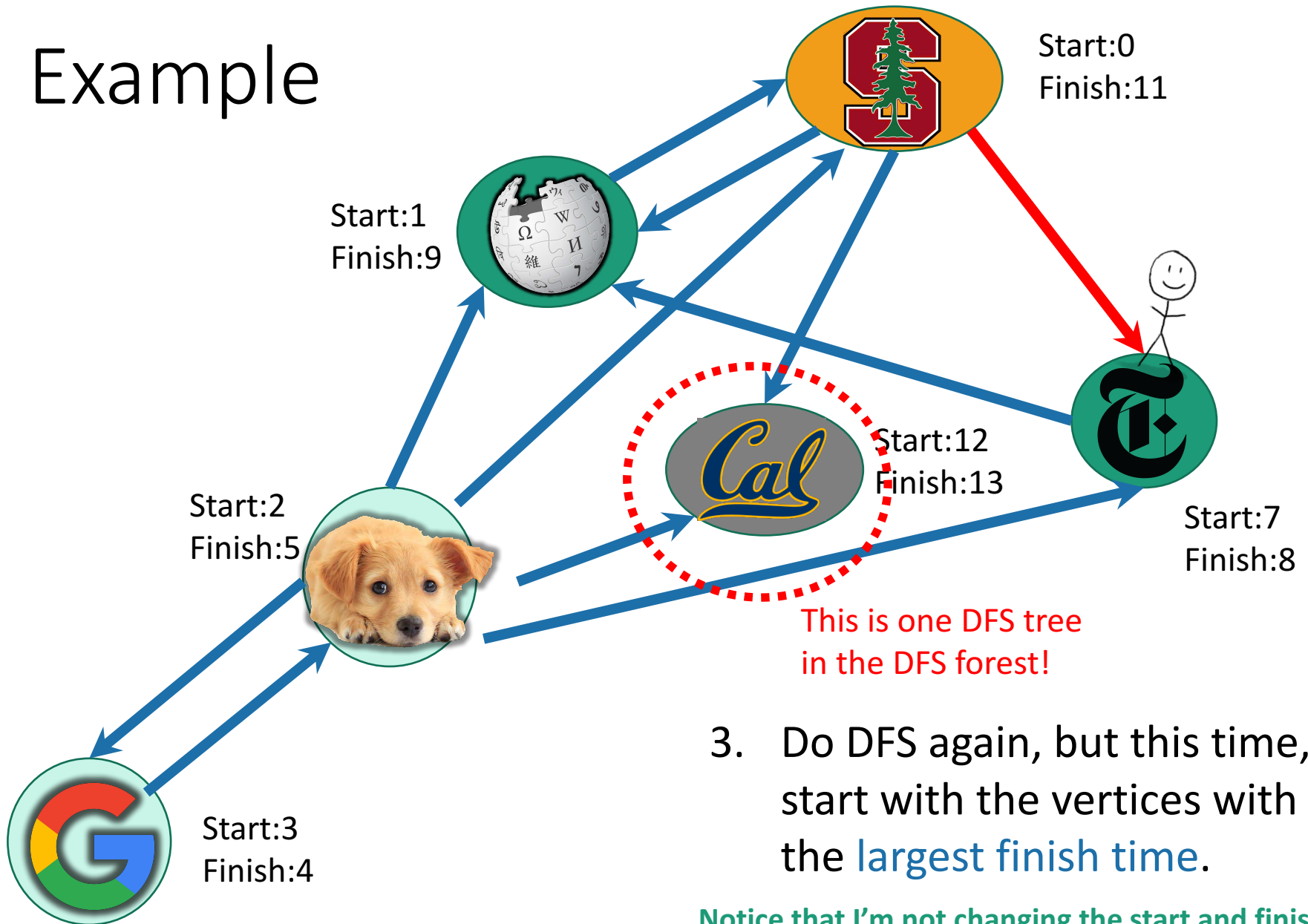
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

Example



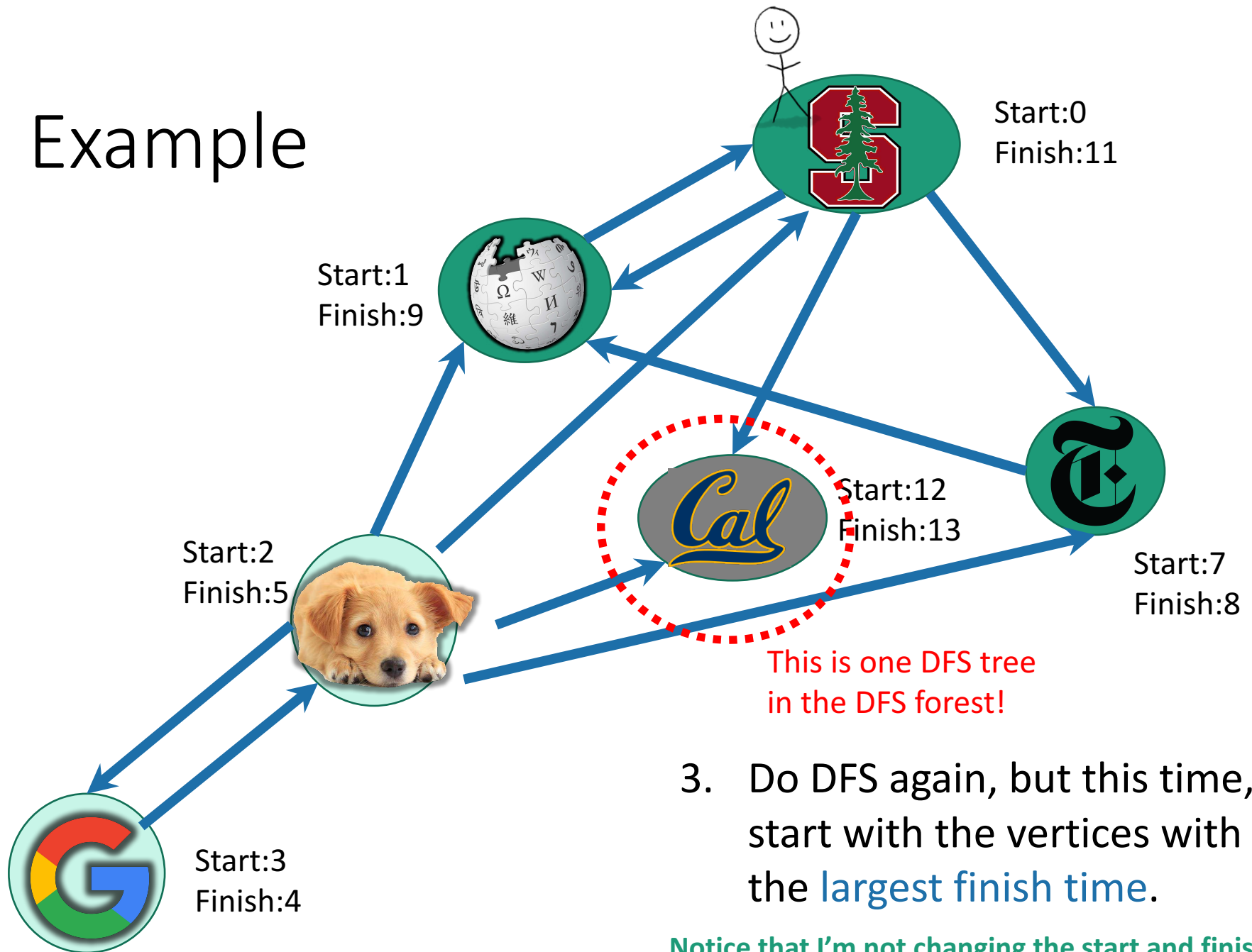
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

Example



Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

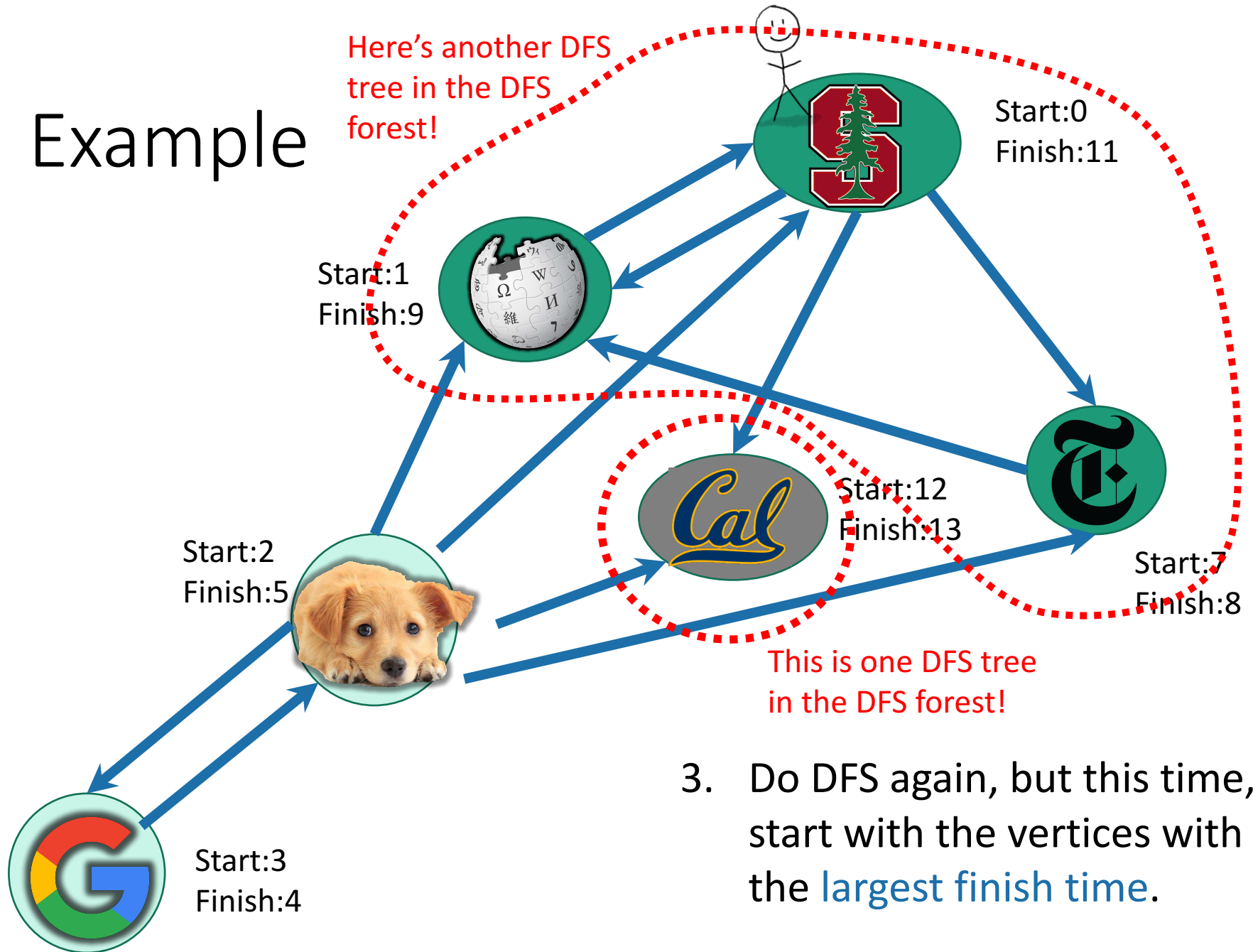
Example



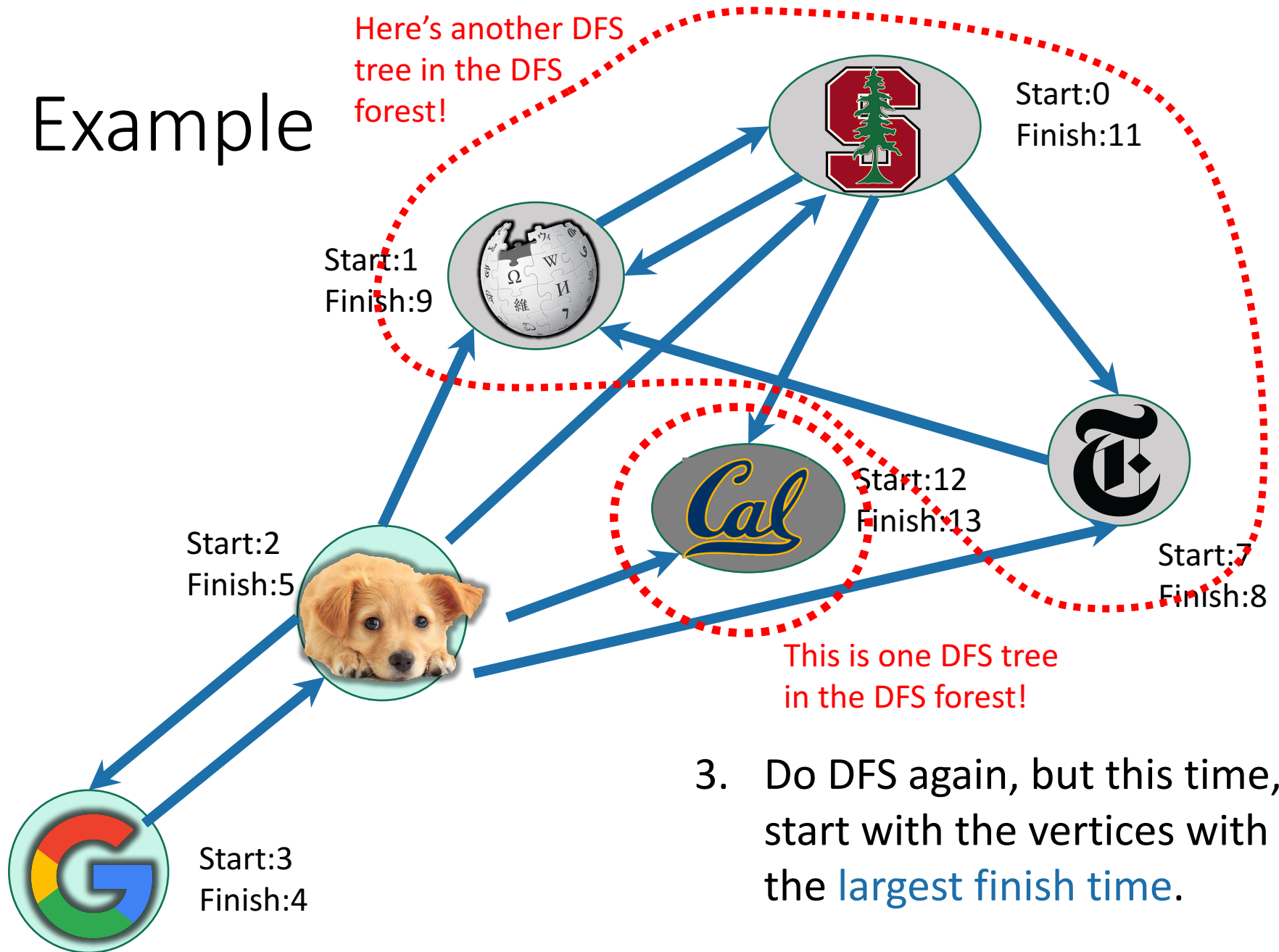
3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

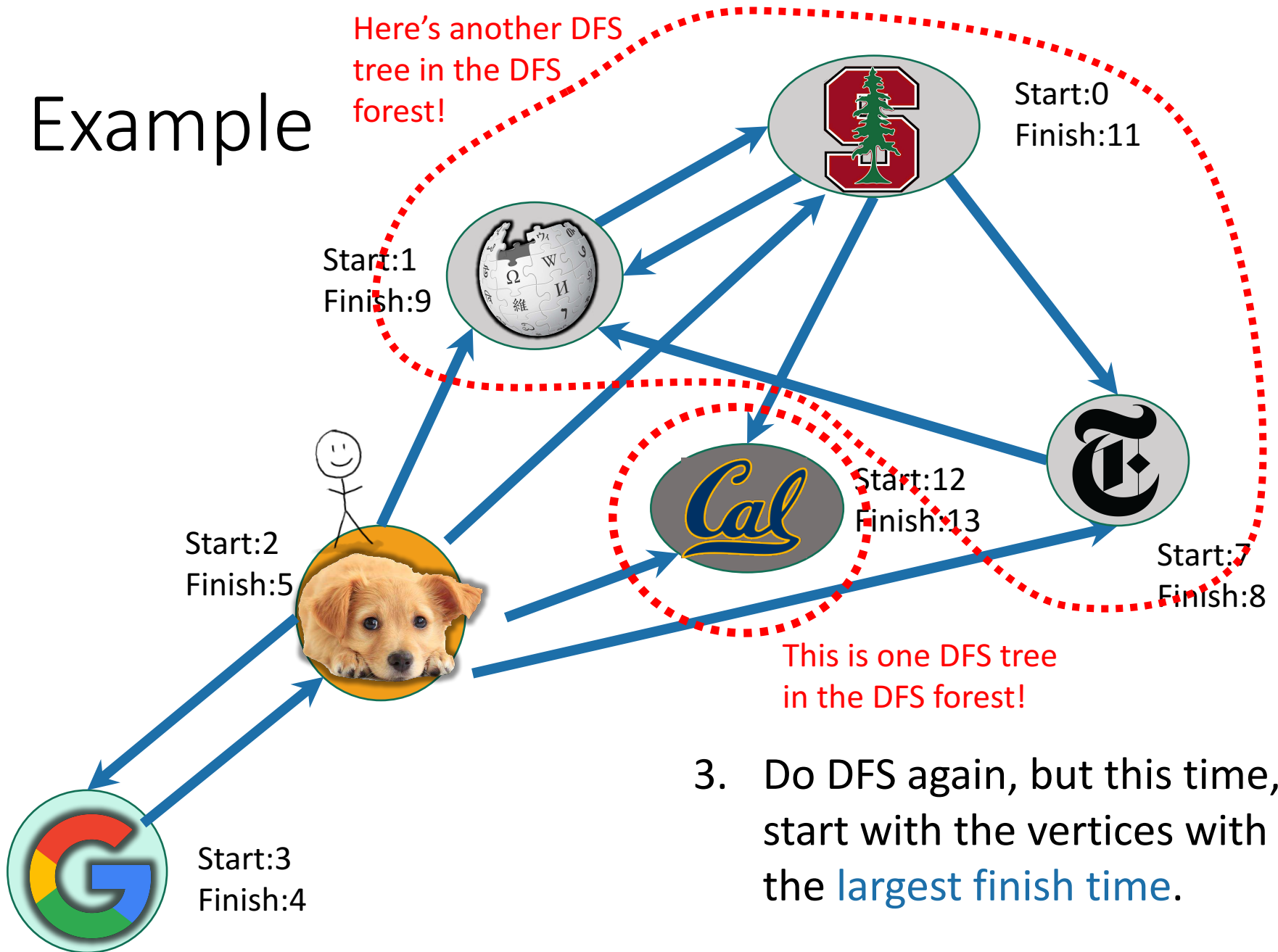
Example



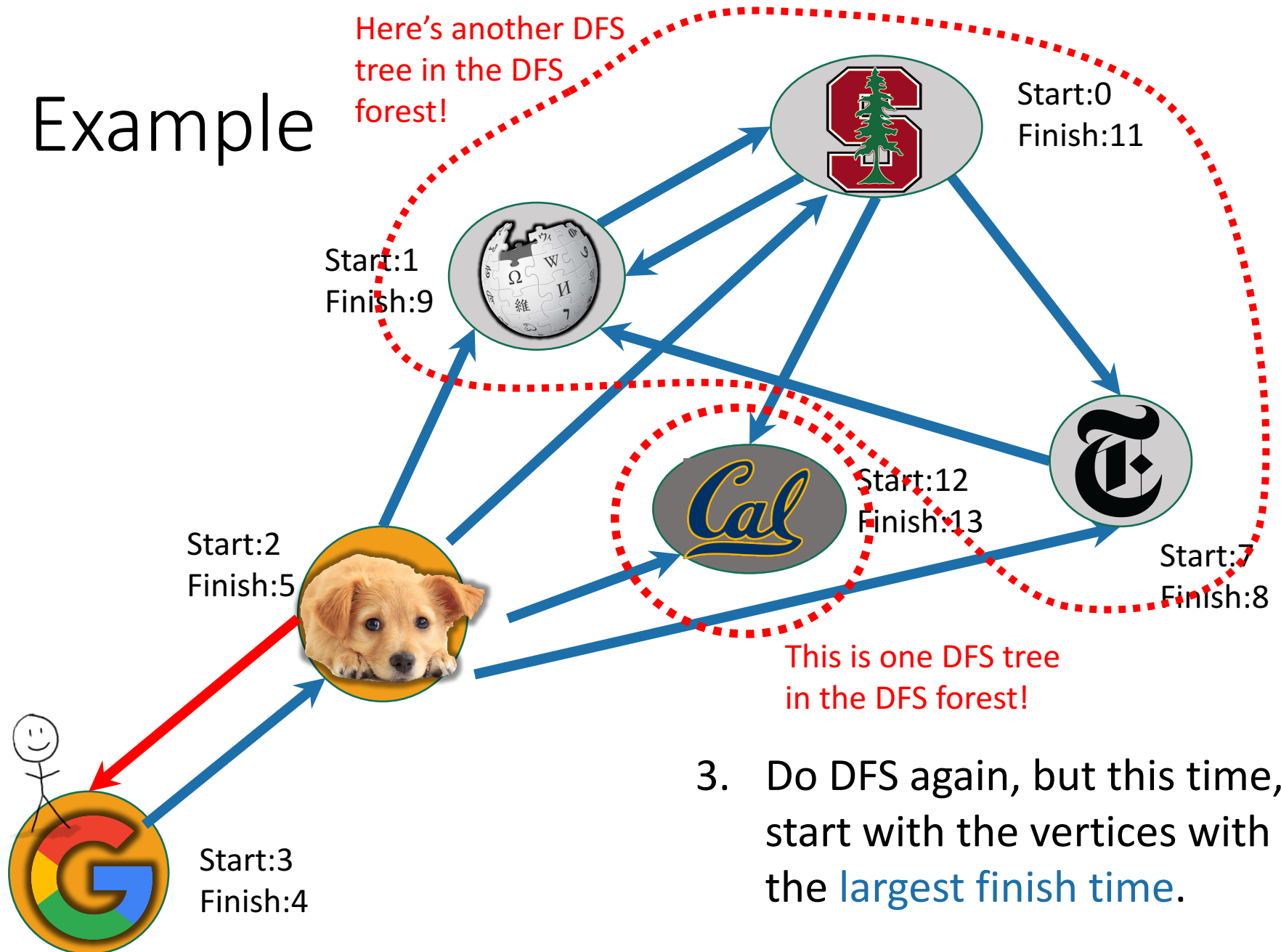
Example



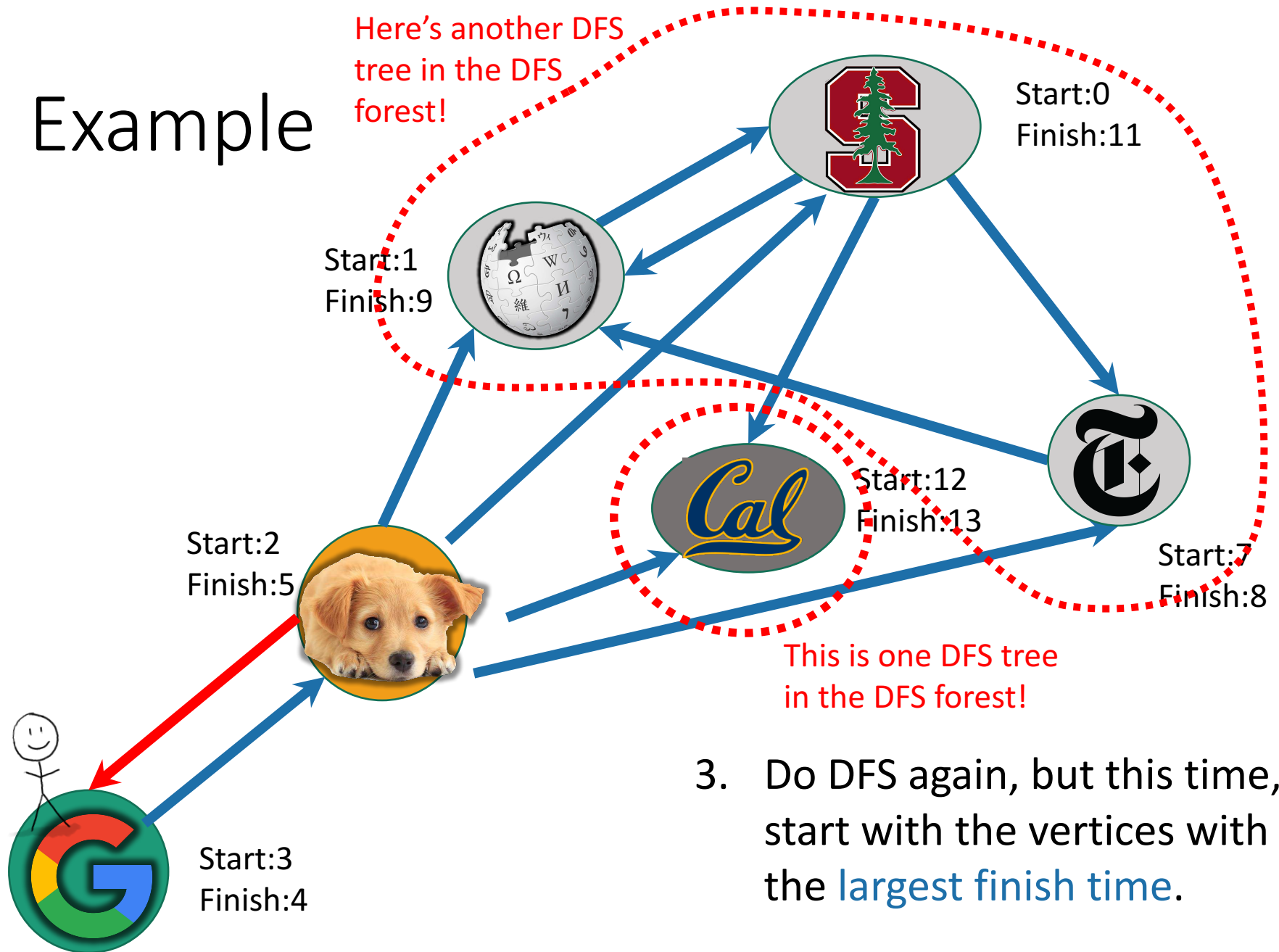
Example



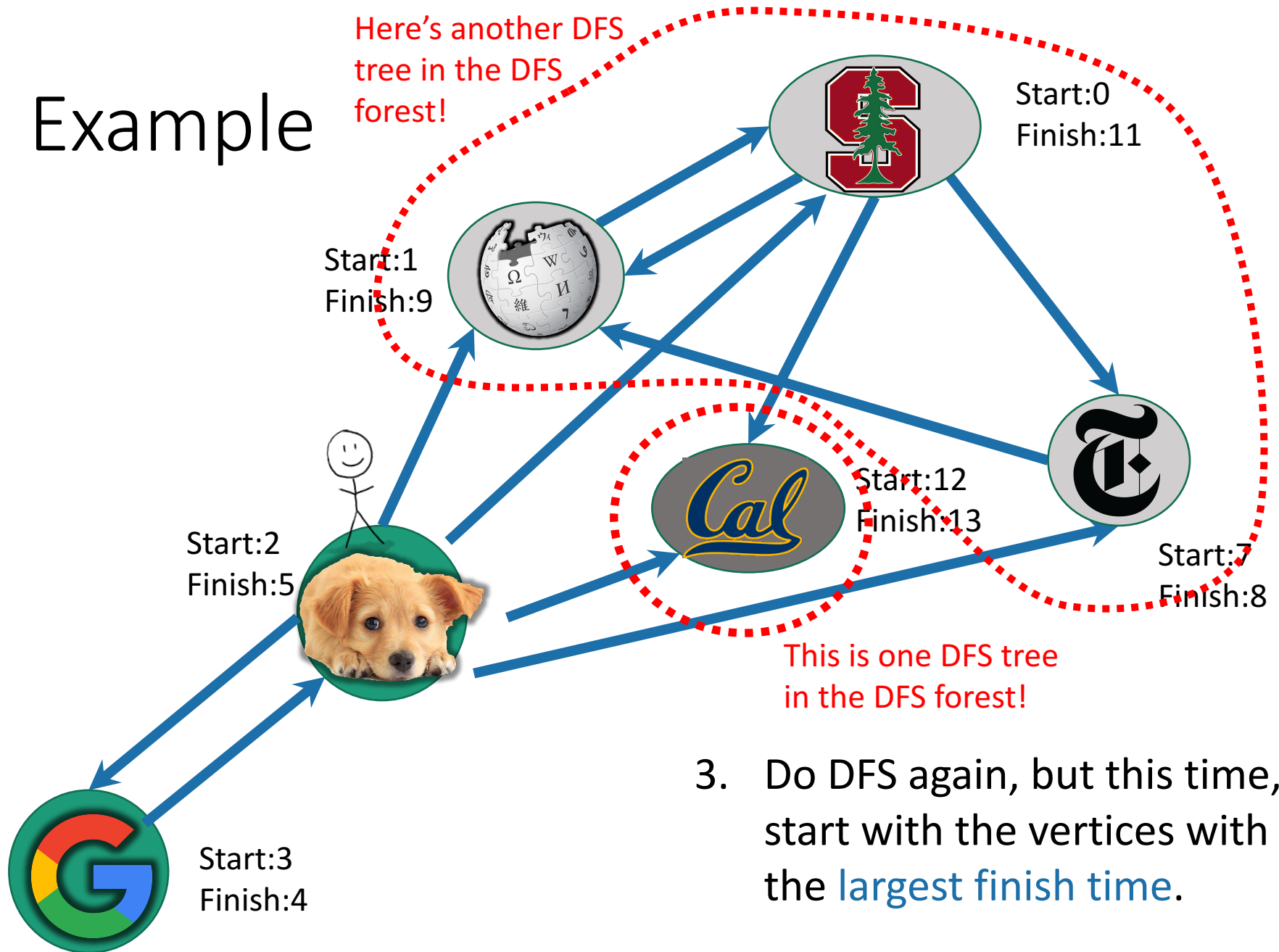
Example



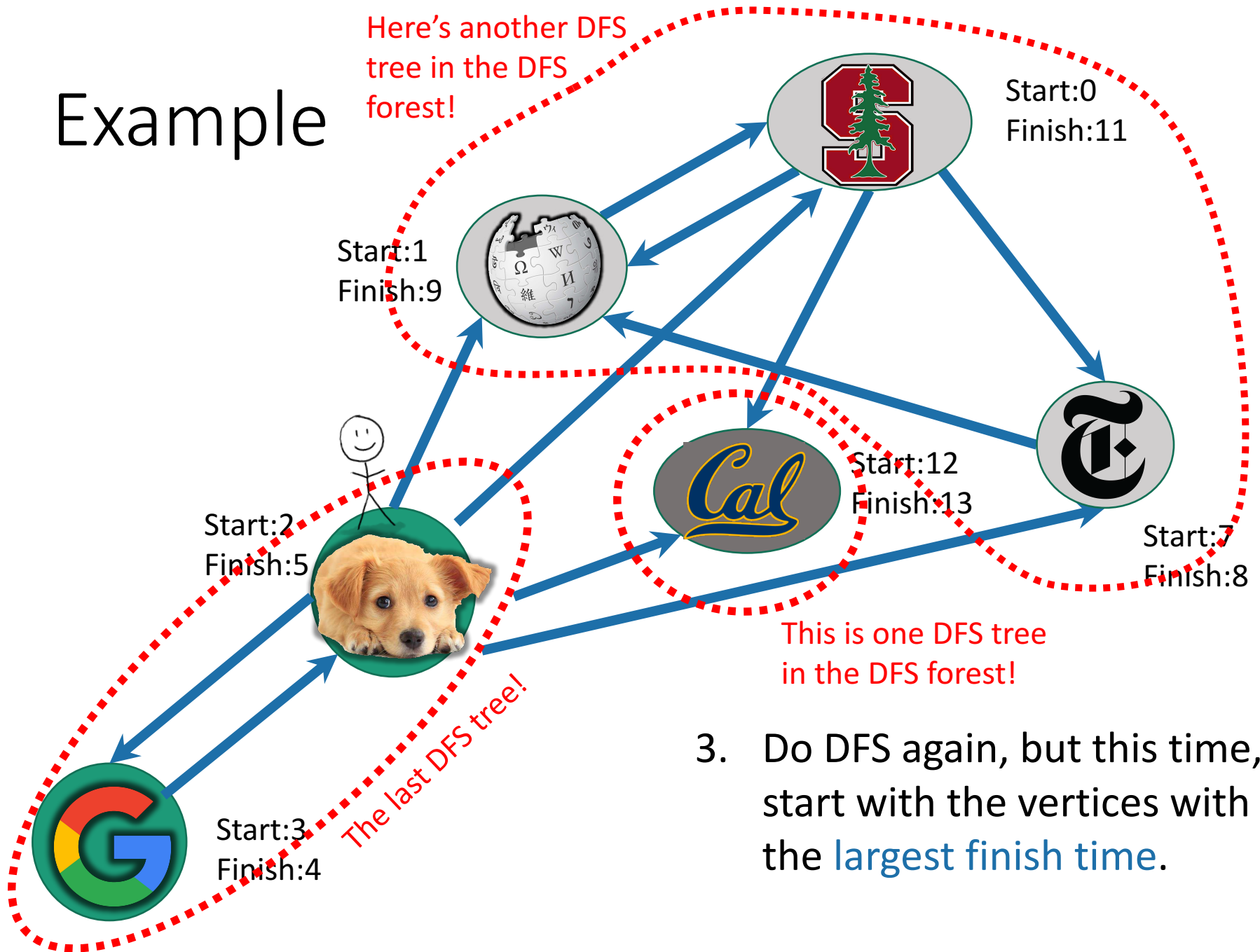
Example



Example

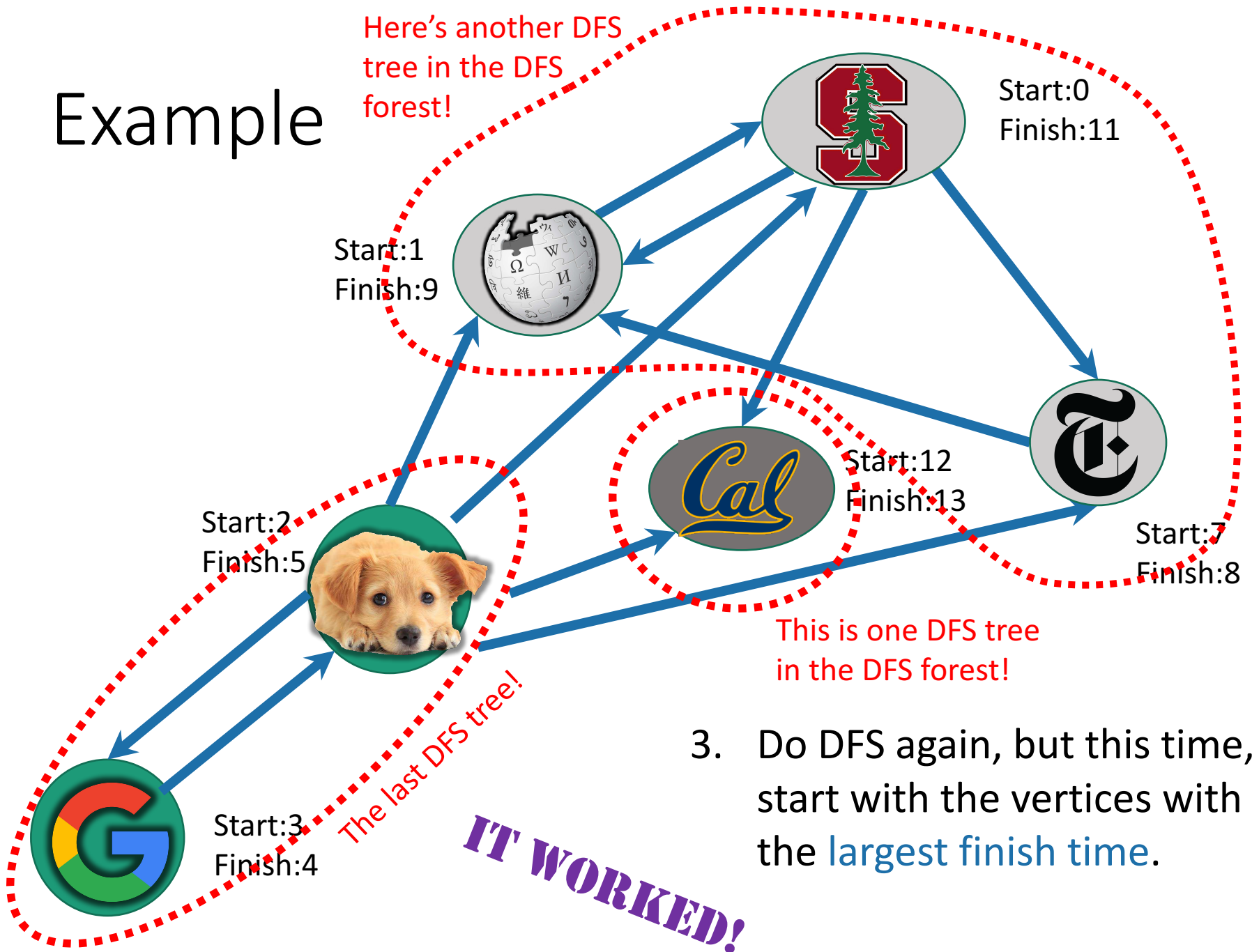


Example



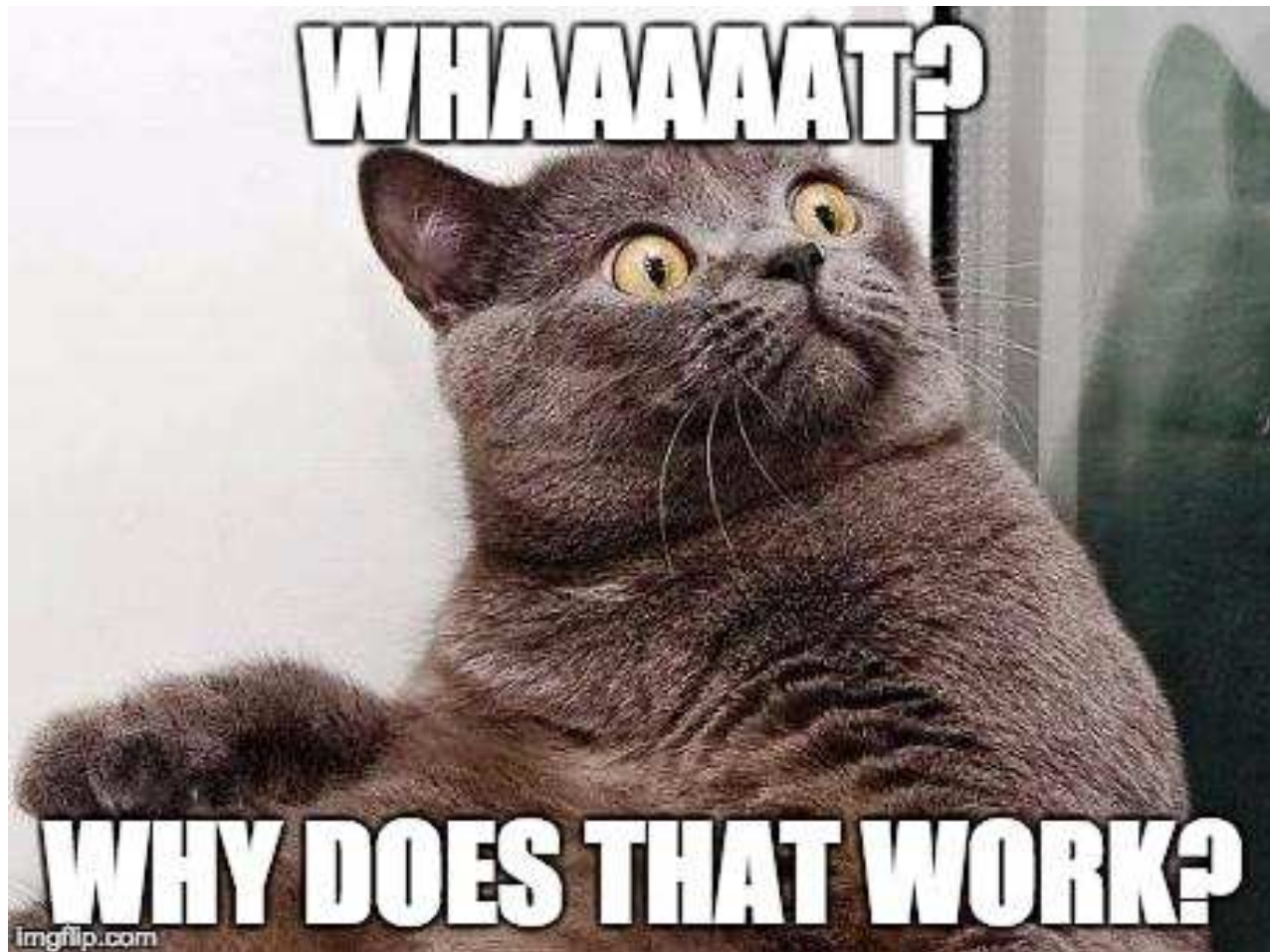
Example

Here's another DFS tree in the DFS forest!

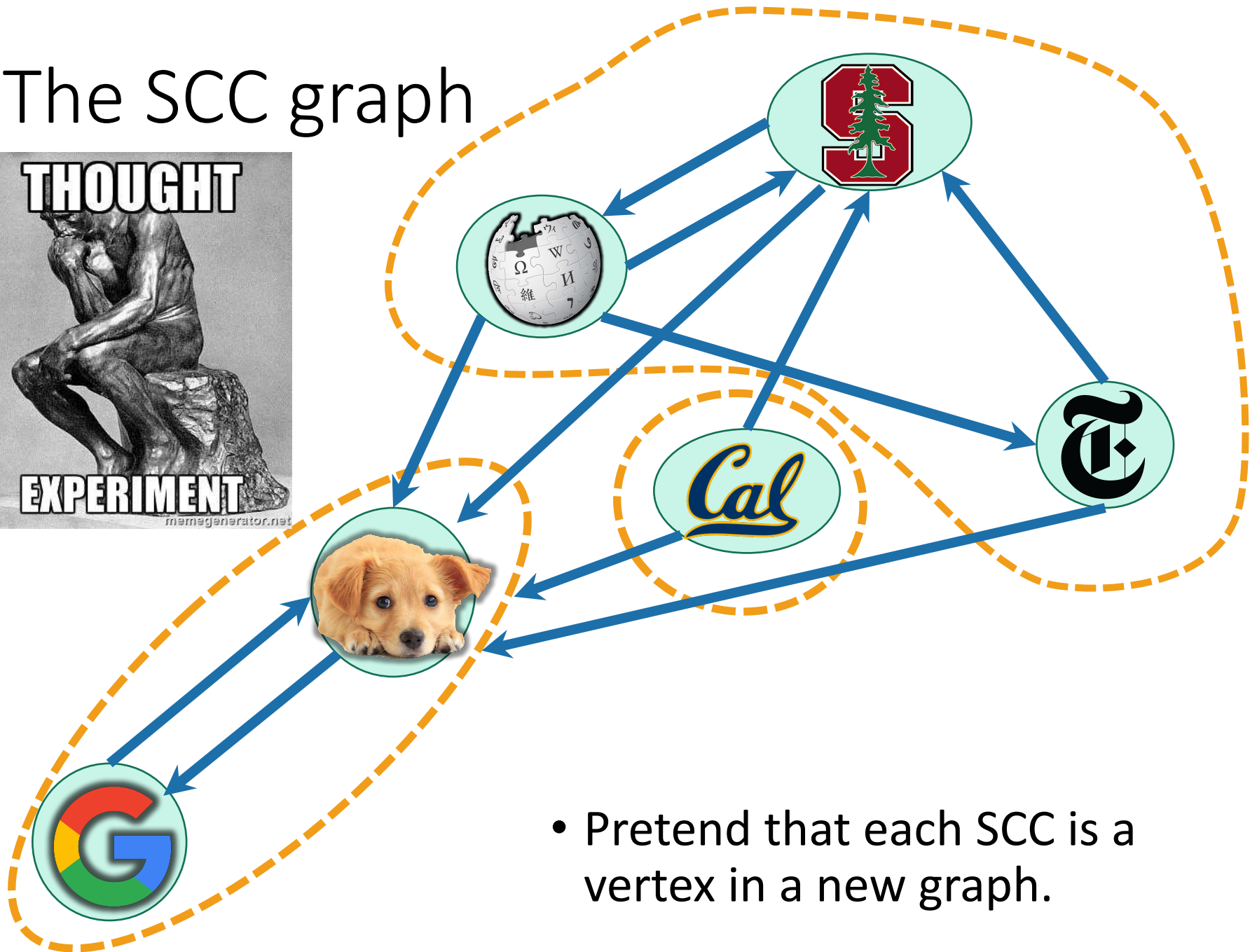
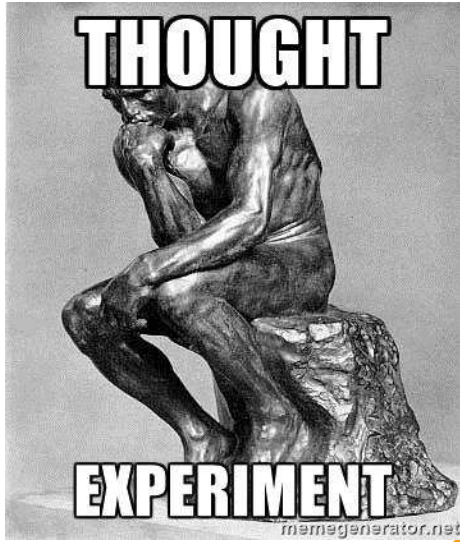


3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

One question



The SCC graph

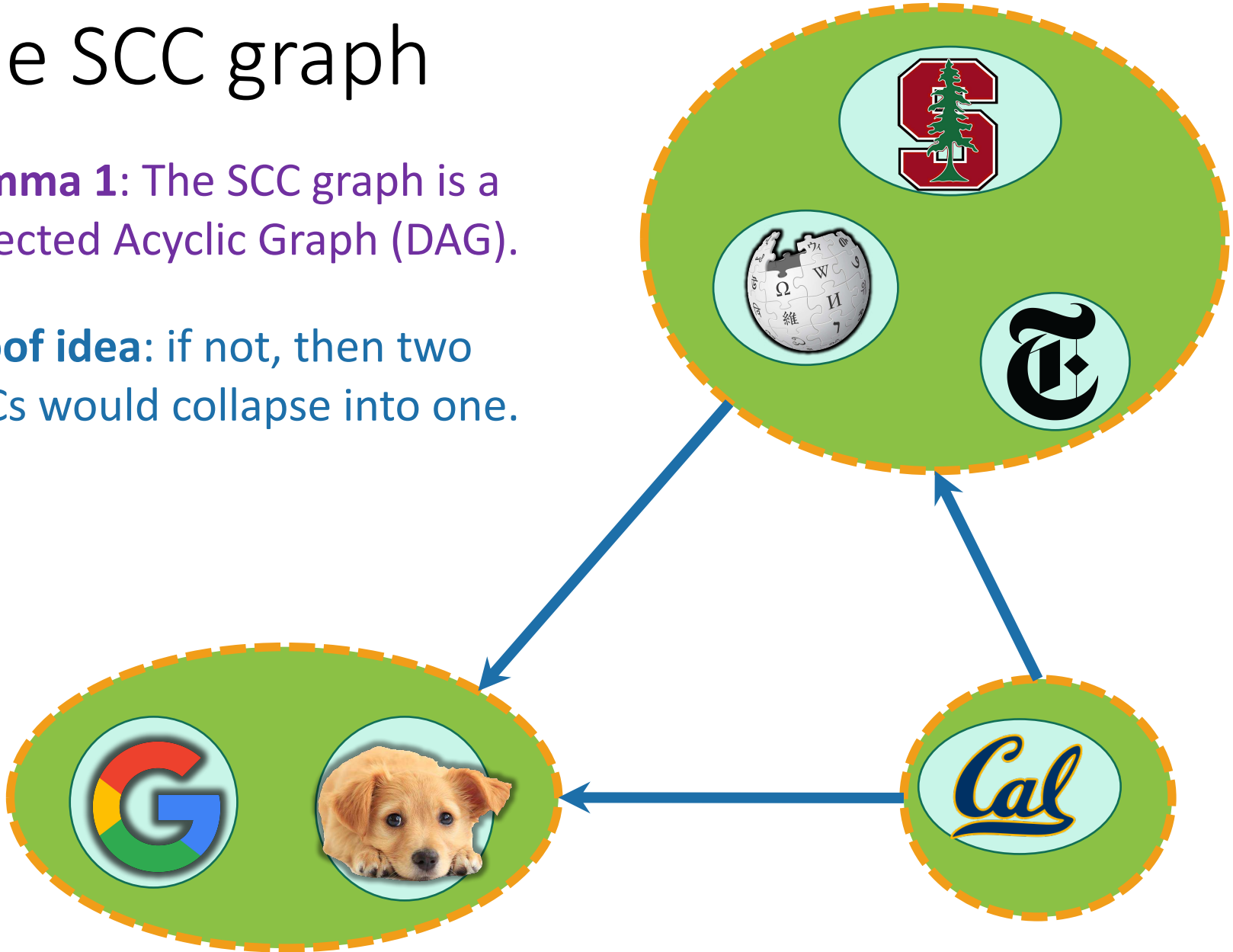


- Pretend that each SCC is a vertex in a new graph.

The SCC graph

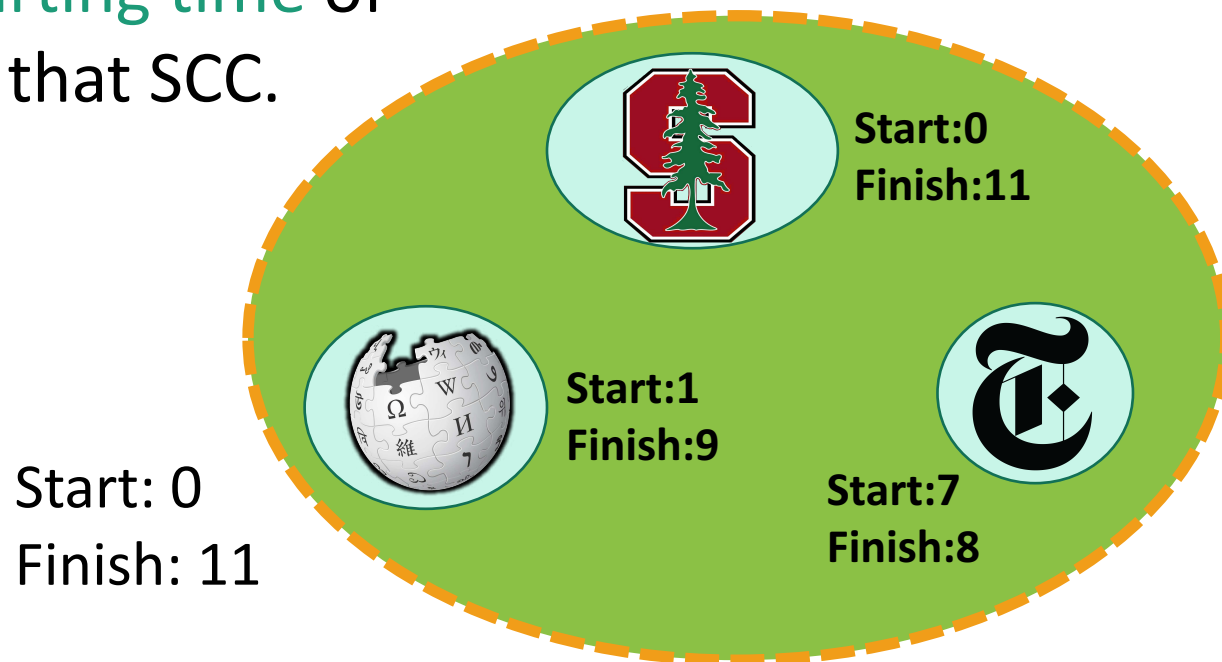
Lemma 1: The SCC graph is a Directed Acyclic Graph (DAG).

Proof idea: if not, then two SCCs would collapse into one.



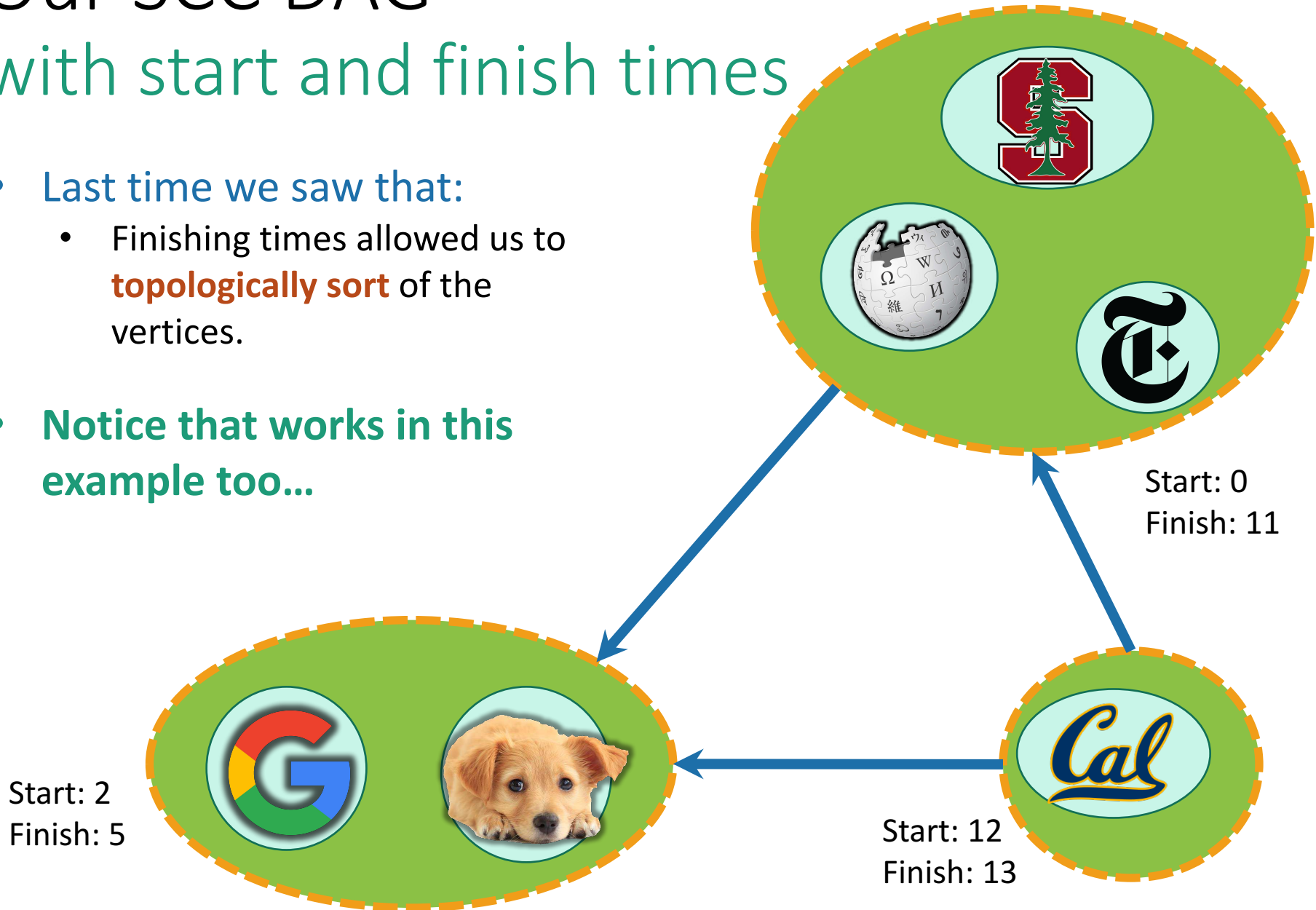
Starting and finishing times in a SCC

- The **finishing time** of a SCC is the **largest finishing time** of any element of that SCC.
- The **starting time** of a SCC is the **smallest starting time** of any element of that SCC.



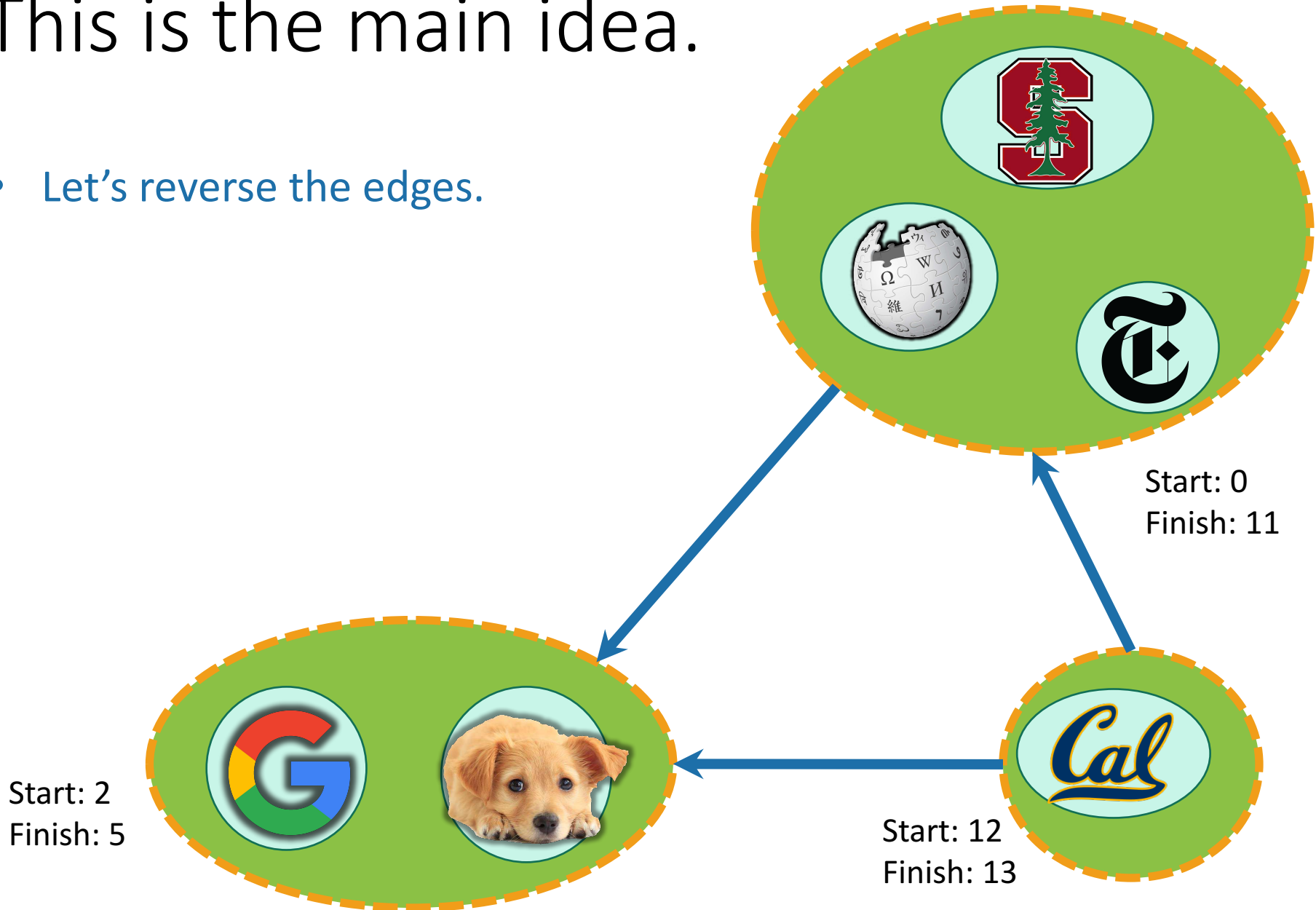
Our SCC DAG with start and finish times

- Last time we saw that:
 - Finishing times allowed us to **topologically sort** of the vertices.
- Notice that works in this example too...



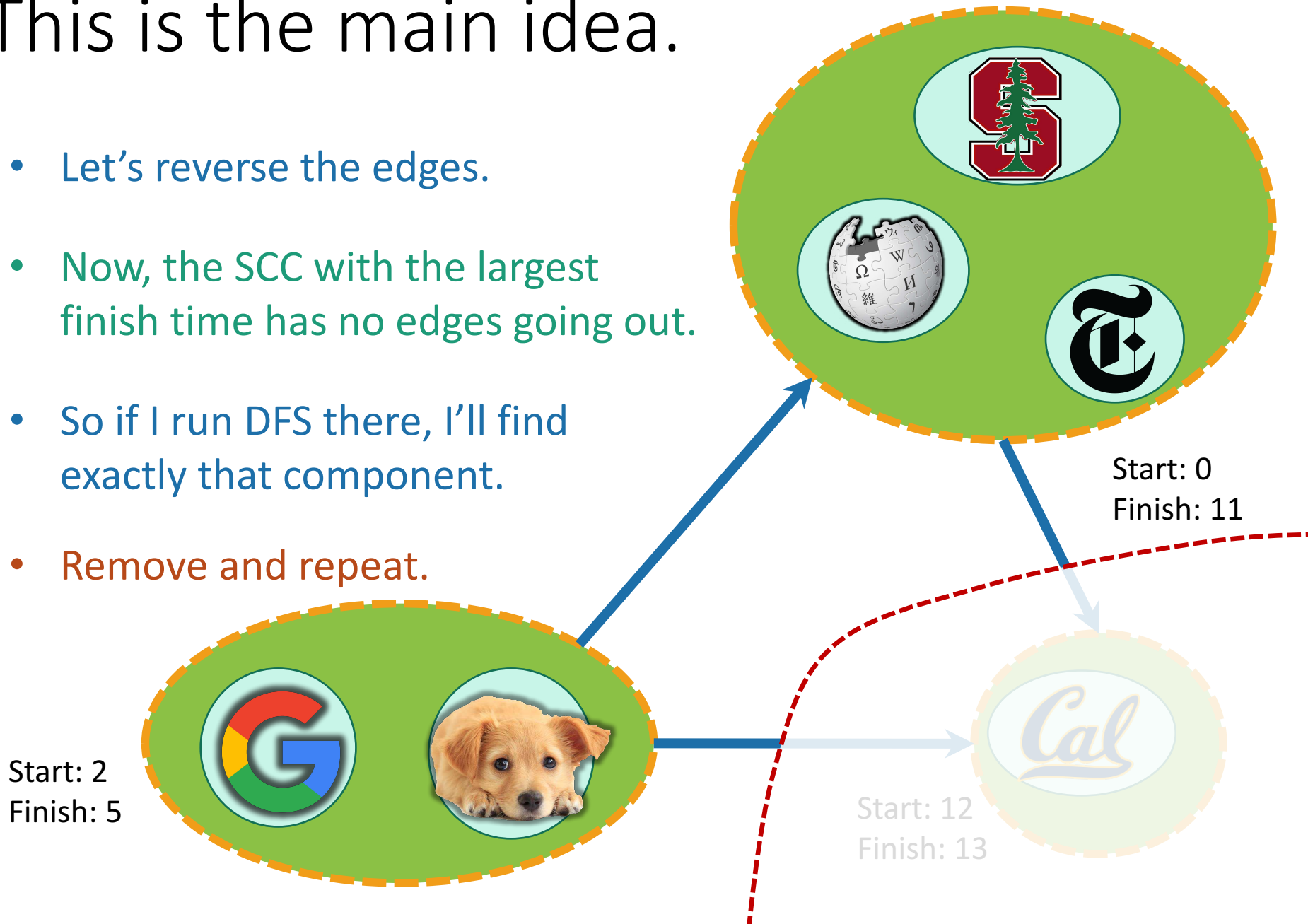
This is the main idea.

- Let's reverse the edges.



This is the main idea.

- Let's reverse the edges.
- Now, the SCC with the largest finish time has no edges going out.
- So if I run DFS there, I'll find exactly that component.
- Remove and repeat.



Let's make this idea formal.

Remember this slide from
Monday...

A more general statement

(this holds even if there are cycles)

This is called the “parentheses theorem” in CLRS

(check this statement carefully!)



- If v is a descendent of w in this tree:



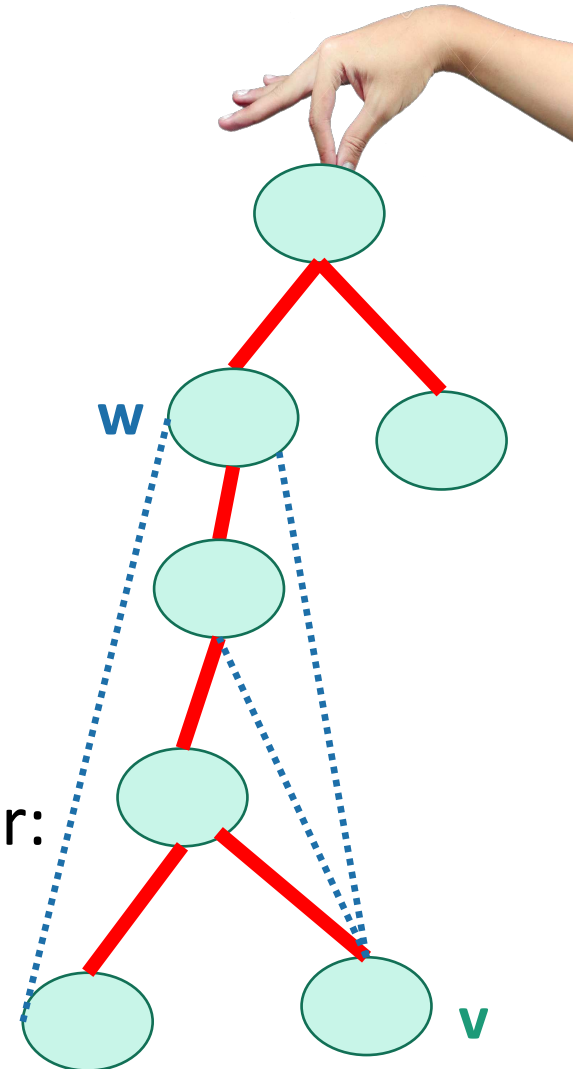
- If w is a descendent of v in this tree:



- If neither are descendants of each other:

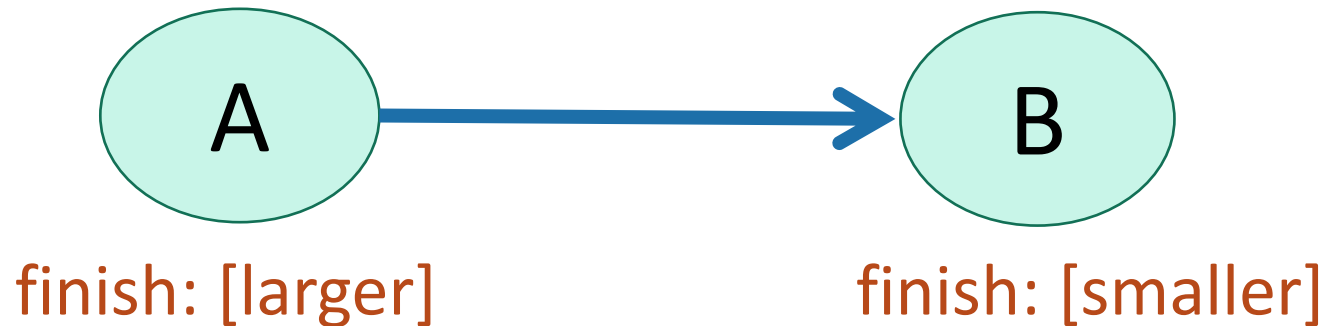


(or the other way around)



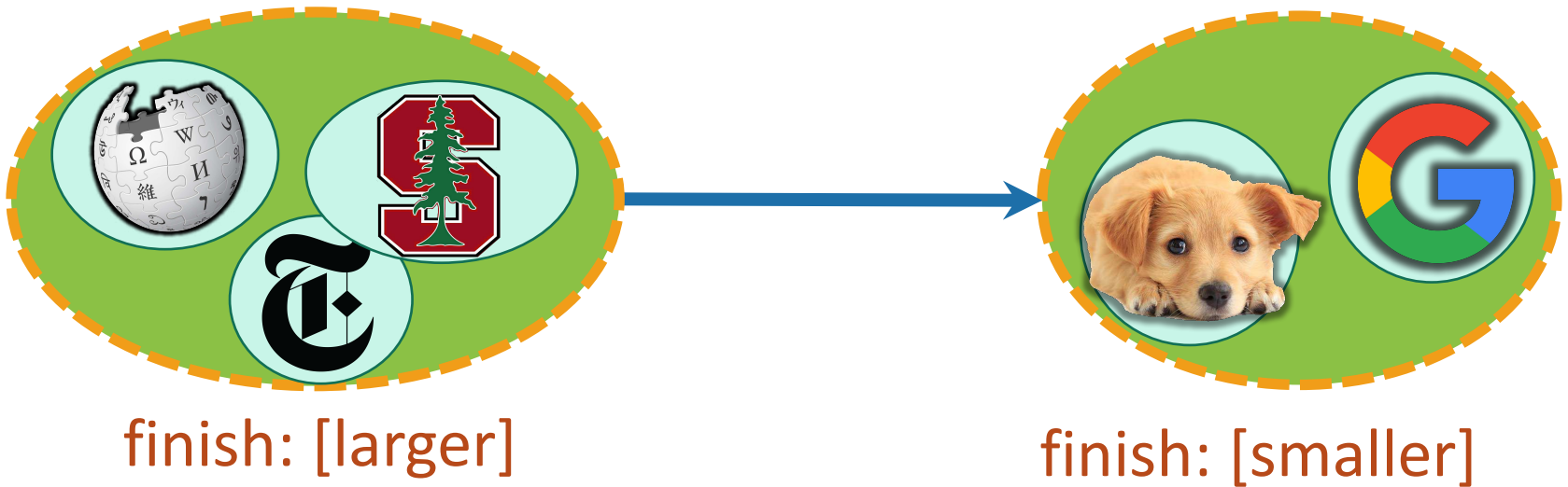
And remember that it implied...

Claim: In a DAG, we'll always have:

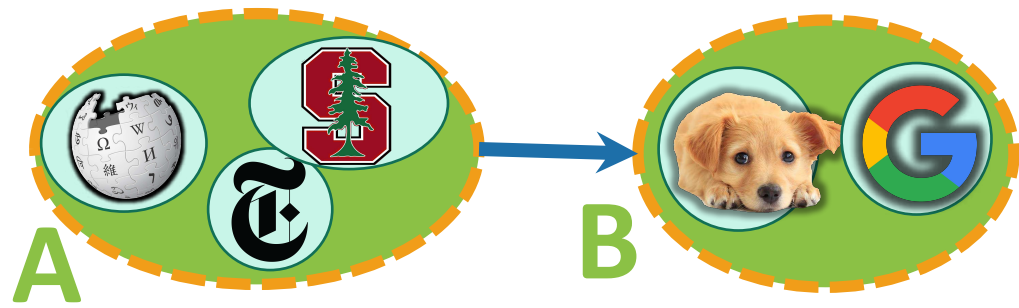


Same thing, in the SCC DAG.

- **Claim:** we'll always have



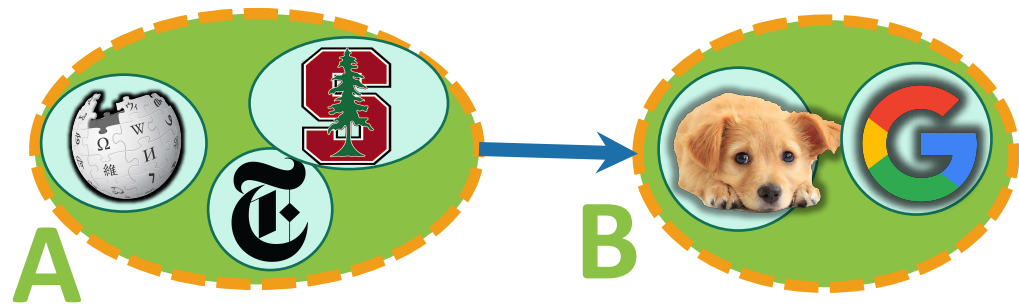
Proof idea



Want to show $A.\text{finish} > B.\text{finish}$.

- **Two cases:**
 - We reached **A** before **B** in our first DFS.
 - We reached **B** before **A** in our first DFS.

Proof idea



Want to show $A.\text{finish} > B.\text{finish}$.

- **Case 1:** We reached **A** before **B** in our first DFS.

- Say that:

- **x** has the largest finish time in **A**;
- **y** has the largest finish in **B**;
- **z** was discovered first in **A**;

So $A.\text{finish} = x.\text{finish}$
 $B.\text{finish} = y.\text{finish}$
 $x.\text{finish} \geq z.\text{finish}$

aka,
 $A.\text{finish} > B.\text{finish}$

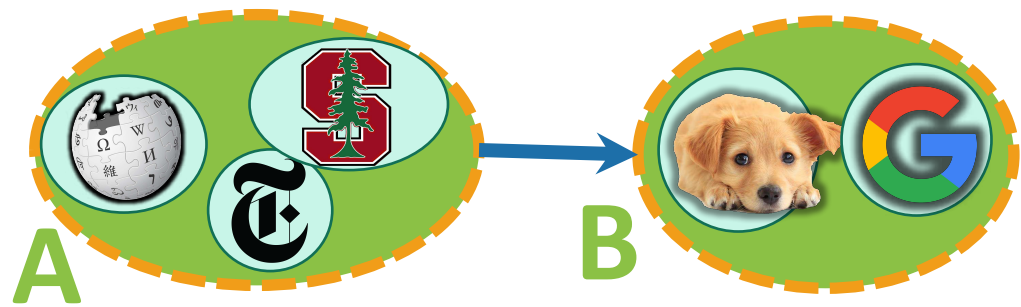
- Then **y** will be discovered in this DFS search, so it must be a descendant of **z** in the DFS forest.



- Then

(by the parentheses theorem)

Proof idea

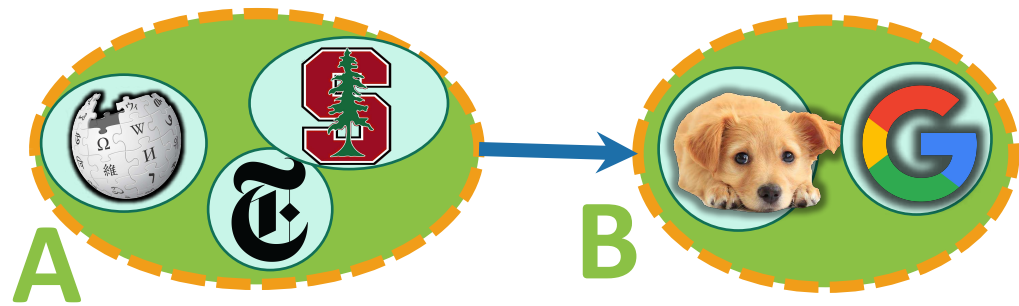


Want to show $A.\text{finish} > B.\text{finish}$.

- **Case 2:** We reached **B** before **A** in our first DFS.
- There are no paths from B to A
 - because the SCC graph has no cycles
- So we completely finish exploring B and never reach A.
- A is explored later after we restart the DFS.

aka,
 $A.\text{finish} > B.\text{finish}$

Proof idea



Want to show $A.\text{finish} > B.\text{finish}$.

- **Two cases:**
 - We reached **A** before **B** in our first DFS.
 - We reached **B** before **A** in our first DFS.
- In either case:

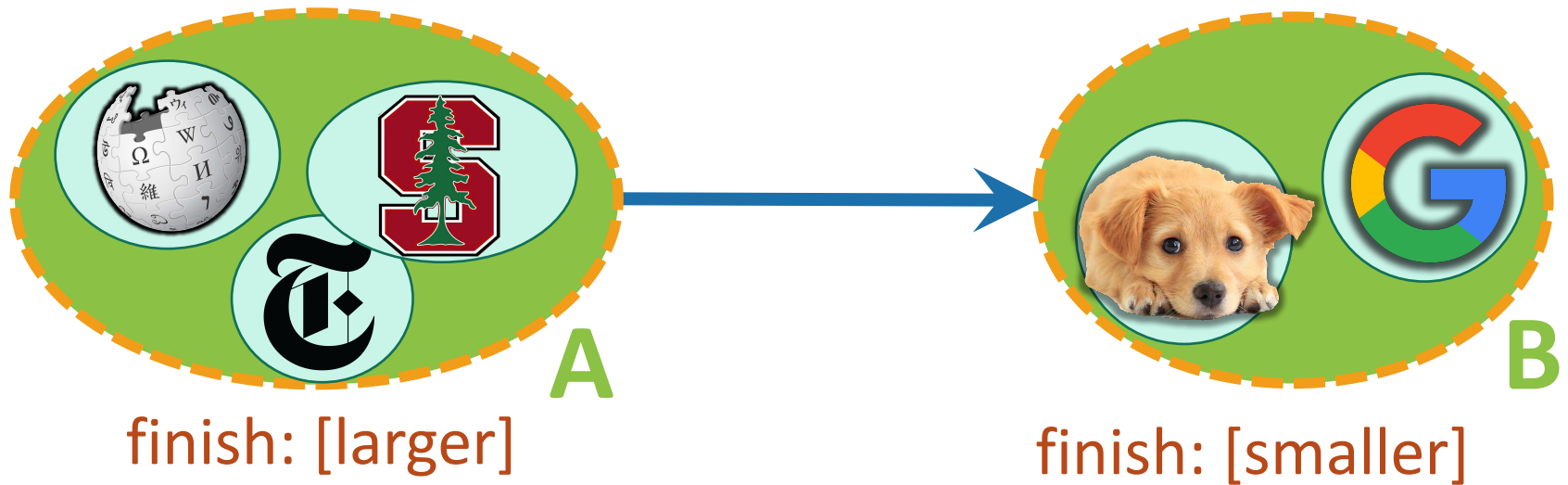
$A.\text{finish} > B.\text{finish}$

which is what we wanted to show.



This establishes:
Lemma 2

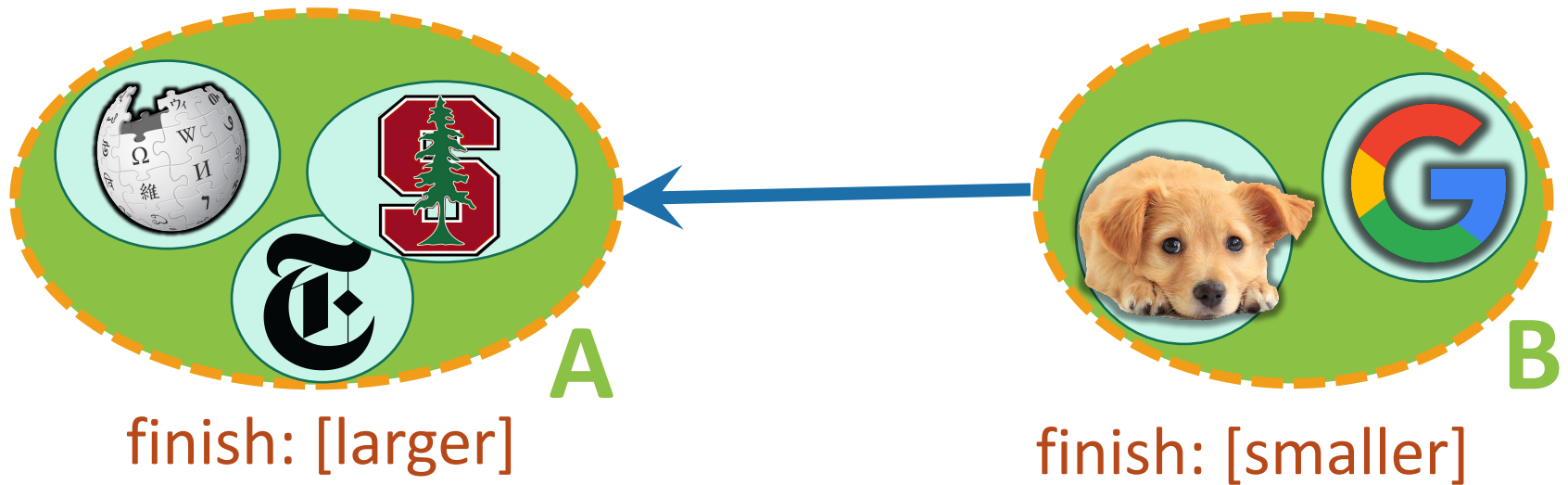
- If there is an edge like this:



- Then $A.\text{finish} > B.\text{finish}$.

This establishes:
Corollary 1

- If there is an edge like this in the **reversed graph**:



- Then $A.\text{finish} > B.\text{finish}$.

Now we see why this works.

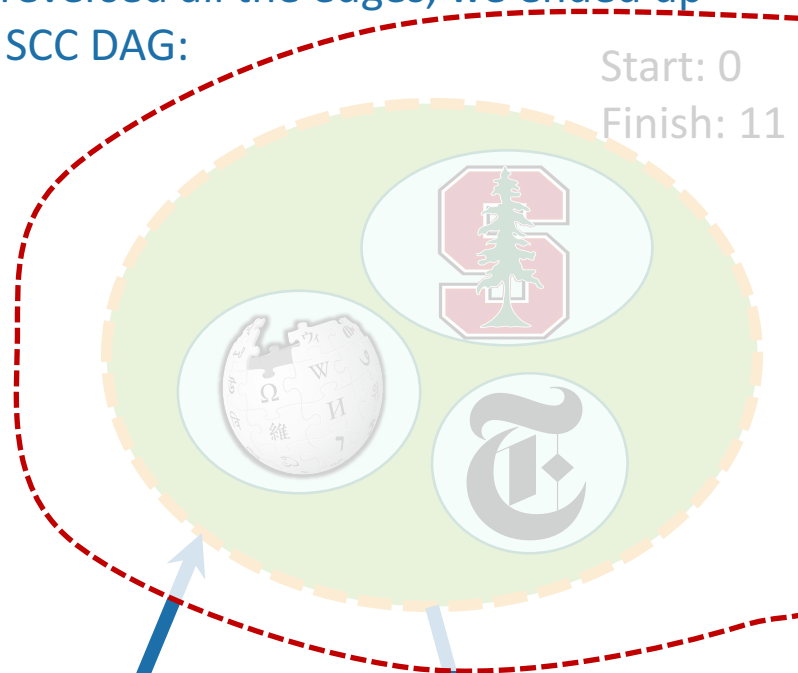
Remember that after the first round of DFS, and after we reversed all the edges, we ended up with this SCC DAG:

- The **Corollary** says that **all blue arrows point towards larger finish times**.
- So if we start with the largest finish time, **all blue arrows lead in**.
- Thus, **that connected component, and only that connected component**, are reachable by the second round of DFS
- Now, we've deleted that first component.
- The next one has the **next biggest finishing time**.
- So **all remaining blue arrows lead in**.
- Repeat.

Start: 2
Finish: 5



Start: 0
Finish: 11



Start: 12
Finish: 13



Formally, we prove it by induction

- **Theorem:** The algorithm we saw before will correctly identify strongly connected components.
- **Inductive hypothesis:**
 - The first t trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
 - Moreover, what's left unvisited after these t trees have been explored is a DAG on the un-found SCCs.
- **Base case:**
 - Trivial for $t = 0$.
 - Moreover, Lemma 1.

Inductive step [drawing on board to supplement]

- **Assume by induction that the first t trees are the last-finishing SCCs, and the rest is still a DAG.**
 - It makes sense to talk about “remaining SCCs;” they form a DAG.
- Consider the $(t+1)^{\text{st}}$ tree produced, suppose the root is x .
- Suppose that x lives in the SCC A .
- Then $A.\text{finish} > B.\text{finish}$ for all remaining SCCs B .
 - This is because we chose x to have the largest finish time.
- Then there are no edges leaving A in the remaining SCC DAG.
 - This follows from the Corollary.
- Then DFS started at x recovers exactly A .
 - It doesn't recover any more since nothing else is reachable.
 - It doesn't recover any less since A is strongly connected.
 - (Notice that we are using that A is still strongly connected when we reverse all the edges).
- **So the $(t+1)^{\text{st}}$ tree is the SCC with the $(t+1)^{\text{st}}$ biggest finish time.**

Formally, we prove it by induction

- **Theorem:** The algorithm we saw before will correctly identify strongly connected components.
- **Inductive hypothesis:**
 - The first t trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
 - Moreover, what's left unvisited after these t trees have been explored is a DAG on the un-found SCCs.
- **Base case:** *[done]*
- **Inductive step:** *[done]*
- **Conclusion:** The second (reversed) DFS forest contains all the SCCs as its trees!

Punchline:
we can find SCCs in time $O(n + m)$

Algorithm:

- Do DFS to create a **DFS forest**.
 - Choose starting vertices in any order.
 - Keep track of finishing times.
- Reverse all the edges in the graph.
- Do DFS again to create **another DFS forest**.
 - This time, order the nodes in the reverse order of the finishing times that they had from the first DFS run.
- The SCCs are the different trees in the **second DFS forest**.



(Clearly it wasn't obvious
since it took all class to do!
But hopefully it is less
mysterious now.)

Recap

- Depth First Search reveals a very useful structure!
 - We saw Monday that this structure can be used to do **Topological Sorting** in time $O(n+m)$
 - Today we saw that it can also find **Strongly Connected Components** in time $O(n + m)$!
 - This was pretty non-trivial.

Next Week

- Weighted graphs!